



275.72

Рейтинг

TINKOFF

IT's Tinkoff — просто о сложном



Azon

6 фев в 18:00

Кто ты, SwiftData

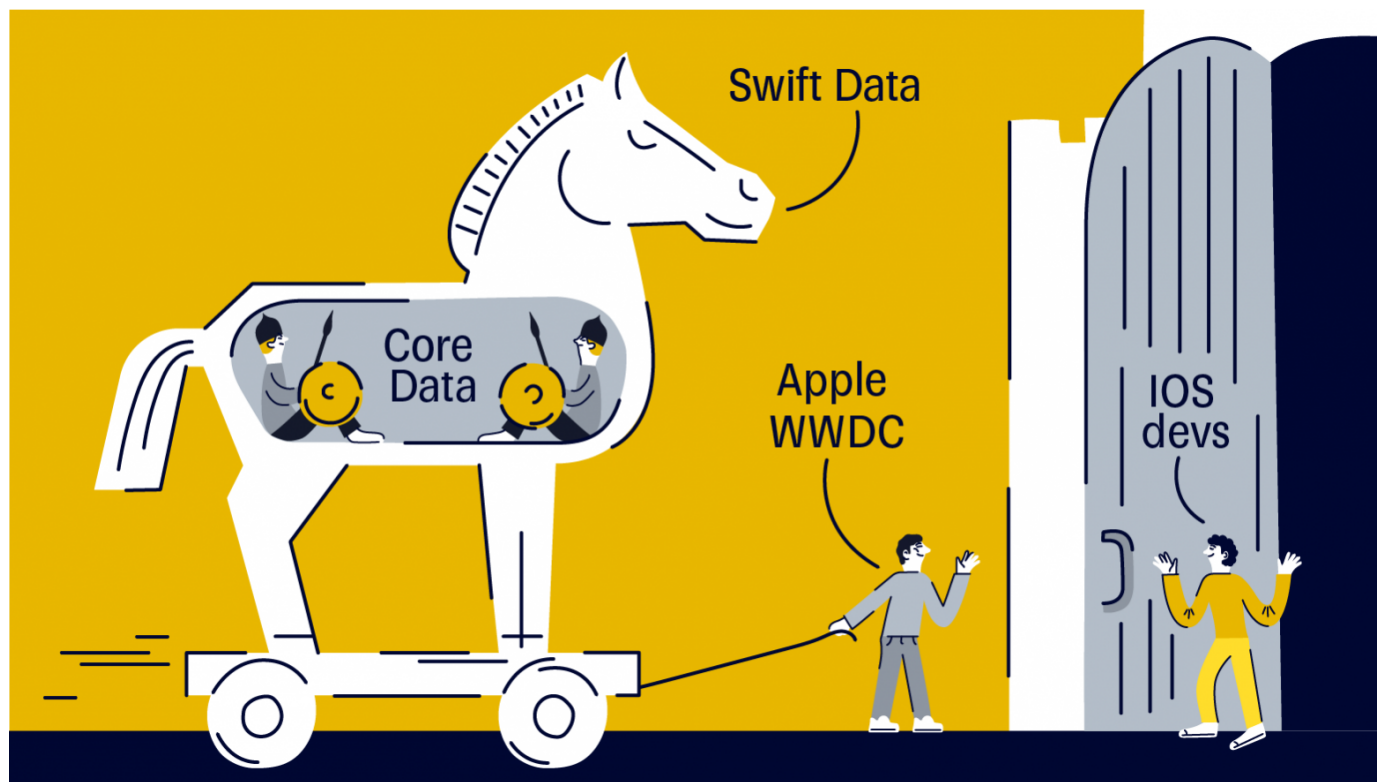
Средний

15 мин

2.5K

Блог компании TINKOFF, Разработка под iOS*, Разработка мобильных приложений*, Swift*, Разработка под macOS

Мнение



Привет! Я Андрей Зонов, стафф-инженер в Тинькофф и большой фанат CoreData. Моя любовь в CoreData началась на старте карьеры, когда я попал в первый Enterprise-проект. Это были времена iOS 4.3., CoreData не имела parent-контекстов и методов perform. Как-то так сложилось, что за свою карьеру я много фиксил классические проблемы в CoreData, и с появлением SwiftData мне стало интересно, остались ли проблемы в SwiftData и что нам дает этот фреймворк.

↑ +12 ↓

24



8

В статье разберем основные концепции и киллер-фичи по SwiftData. Пройдемся по основным отличиям и тому, как можно мигрировать с CoreData на актуальную SwiftData. Копнем внутрь SwiftData, узнаем, как она устроена под капотом, и подведем итоги стоит ли вообще переезжать на новый фреймворк Apple.

Концепция @Model

Основная и первая киллер-фича SwiftData — это переворот концепции. Если в CoreData мы создавали файл `.xcdatamodeld` и рисовали схемки в визуальном интерфейсе, то так же как и со SwiftUI, в SwiftData появляется единый источник правды — это код.

Определяем схему Entity прямо в коде:

```
final class Post {  
  
    var id: String  
    var message: String  
    var timestamp: Date  
    var imageURLs: [URL]
```

Больше не нужно думать об архитектуре нашего компонента — нужно ему хранилище или нет. Если на слое модели уже есть какой-то класс, например класс публикации `Post`, то нужно просто добавить макрос `@model`, чтобы сделать класс персистентным и добавить возможность сохранения его в `Storage`.

Кажется, что именно макросов ждала CoreData со времен iOS 5. Потому что макросы дают ту самую `compile-time-безопасность` и возможность валидации схем, которой не хватало в CoreData.

Но это не все, что умеет SwiftData-модель. Если у исходного класса были ссылки на другие классы, то добавление макроса `@model` создаст связи с сущностями, ссылки на которые были в исходном классе. Например, связь с аттачментами или с автором-пользователем. В случае с автором мы получаем связь типа один к одному, а в случае с аттачментами — один ко многим. Связи устанавливаются из кода:

```
@Model  
final class Post {  
  
    var id: String  
    var message: String  
    var timestamp: Date
```

```
var attachments: [Attachment]
var author: User
```

В коде появились новые макросы.

Атрибут Unique. Если добавить его к полю ID, то можно помочь SwiftData не создавать дублирующую запись, а найти ту, что есть, и обновить поля, которые не совпадают. То есть при добавлении новой публикации в Store, если уже в Store есть публикация с идентичным ID, SwiftData сделает не insert, а upsert.

```
@Model
final class Post {
    @Attribute(.unique)
    var id: String
    var message: String
    var timestamp: Date

    var attachments: [Attachment]
    var author: User
```

Макрос Relationship. Тут вы встретите знакомое из Core Data слово — Cascade. Он позволяет при удалении публикации автоматически удалять все attachments, связанные с ней.

Здесь же можно выставить инверсные зависимости. Если у attachment есть несколько полей, названных или ссылающихся на пост, мы можем подсказать SwiftData, какое именно является инверсной зависимостью.

```
@Model
final class Post {
    @Attribute(.unique)
    var id: String
    var message: String
    var timestamp: Date
    @Relationship(deleteRule: .cascade, inverse: \Attachment.post)
    var attachments: [Attachment]
    var author: User
```

Макрос `Transient`. Apple активно использует этот макрос в реализации самой SwiftData. Если ваш класс модели имеет свойство, которое вы не хотите писать в постоянное хранилище, достаточно аннотировать каждое из этих свойств макросом `@Transient`. Например, это может быть калькулируемое поле, зависящее от сегодняшней даты.

```
@Model
final class Post {
    @Attribute(.unique)
    var id: String
    var message: String
    var timestamp: Date
    @Relationship(deleteRule: .cascade, inverse: \Attachment.post)
    var attachments: [Attachment]
    var author: User
    @Transient
    var isDraft: Bool = false
}
```

Концепция Model Container

Следующая концепция SwiftData — это Model Container или контейнер моделей. Если до этого мы рассмотрели модели, то Model Container — тот самый мостик между схемой и хранилищем. Model Container получает в конструкторе набор моделей, с которыми ему нужно работать.

Model Container достаточно умный, чтобы заметить, что у сущности Post есть связи с attachment и user, такая конфигурация будет валидной. Model Container поймет, что у поста есть связи, и самостоятельно добавит их в схему базы данных.

Контейнер настраивает хранилище, конфигурации и план миграции с предыдущих схем на актуальные. Например, в SwiftData можно настроить in-memory тип хранилища и не хранить персистентность. Можно настроить обратную ситуацию, когда у нас есть persistent store, но мы не можем в него писать — readOnly. Тут же настраиваются такие нюансы, как Cloud Kit или прямой путь к .sqlite-файлу. Контейнер связывает нашу схему с физическим хранилищем на диске.

Кроме контейнера у моделей есть контекст. Контейнер — компаньон контекста. Контекст в SwiftData, в отличие от CoreData, связан именно с контейнером.

```
let container = try ModelContainer(for: Post.self)
let context = container.mainContext
```

```
let newContext = ModelContext(container)
newContext.autosaveEnabled = false

let post = Post(message: "Привет!")
newContext.insert(post)
try newContext.save()
```

Сам контекст мы получаем из контейнера. Apple пыталась это внедрить уже в последних версиях CoreData, но здесь это вышло абсолютно на новый уровень. Для создания контекста нужно передать контейнер, в рамках которого он будет работать. И при создании модели сущности Post мы должны явно вызвать метод insert у контекста и save.

Я пропустил autosave enabled, потому что вся SwiftData работает на дефолтах и по дефолту все контексты автосохраняемы. Если мы выключаем автосохранение у созданного контекста, то не нужно вызывать метод save.

Когда вызывается метод save:

- с Swift UI все очевидно: когда перерисовывается view, проверяется, есть ли изменения в контексте. И если они есть, то триггерится сохранение;
- с UIKit каждый раз, когда приложение UI Application меняет метод своего жизненного цикла, то есть переходит в foreground или в background, вы переключаете приложение или закрываете, явно вызывается save. Мы можем быть уверены, что сохранение будет перед тем, как приложение выгрузится из памяти. Даже если приложение просто висит на экране у пользователя, через какое-то время на RunLoop срабатывает периодический таймер, и если есть изменения, также триггерится save.

Важно, что контекст связан с контейнером, но не с моделью, поэтому в SwiftData возможно создание объекта с контекстом сразу. Модель самостоятельно работает без контекста, и мы можем ее использовать не только на слое модели, конструировать ее в отрыве от контейнера и таким образом не думать о персистентности на ранних этапах конструирования модуля.

Контекст — это своеобразный слепок данных, выгруженных из хранилища. Именно контекст следит за тем, какие данные в хранилище обновляются, возможно с помощью других контекстов, и сообщает об этом интерфейсу. И также при изменении моделей в самом контексте он проводит эти данные в Store. У одного хранилища может быть несколько слепков, но у каждого контекста может быть только одно хранилище. Связь — одно хранилище ко многим контекстам.

На уровне контекста настраивается автосейв, и если автосейв выключен, очевидно, что можно сделать так же, как и в CoreData, undo и redo.

Концепция Predicate

Следующая концепция и киллер-фича SwiftData — это предикаты. Предикаты работают на основе макросов, и они имеют очевидное преимущество над предикатами CoreData — Compile-Time-Safety. Кроме того, предикаты — это pure Swift, который преобразуется в select-запросы. Базовая работа с коллекциями может быть преобразована в select-запрос с Join-ом. Предикаты их поддерживают, но скрывают.

В compile-time предикаты подскажут, возможно ли собрать select-запрос, и если нет, то почему. Такая возможность есть из-за того, что предикаты реализованы через макросы.

У предиката есть компаньон — FetchDescriptor. Он нужен для того, чтобы выполнить fetch-запрос, его название говорит «я описываю fetch-запрос». Мы передаем предикат в FetchDescriptor и после этого передаем дескриптор в контекст.

FetchDescriptor нужен, потому что внутри SQL есть select, where и sortby. Sortby реализованы через key-value-кодирование. Key-value-кодирование опять в Swift-way, безопасный и удобный. FetchDescriptor имеет в конструкторе эти два аргумента — predicate и sortby, и оба аргумента опциональны.

FetchDescriptor может иметь просто пустой конструктор, тогда он получит все записи из нашей таблицы, например для пагинации.

На уровне дескриптора можно настроить исключение тех данных, которые еще не сохранены. И опять же, мы возвращаемся к тому, что у контекстов есть автосейв. Автосейв можно выключить. Если мы выключили автосейв, то, соответственно, мы, возможно, хотим фетчить данные из контекста, но только те, что уже были сохранены, и исключать те, что не были сохранены.

```
let today = Date()
let kittyFeed = #Predicate<Post> {
    $0.timestamp < today &&
    $0.message.contains("котики")
}
var descriptor = FetchDescriptor<Post>(
    predicate: kittyFeed,
    sortBy: [SortDescriptor(\.timestamp)]
)
descriptor.includePendingChanges = false

let posts = try newContext.fetch(descriptor)
```

Например, есть публикация. Мы не хотим сохранять публикацию до того, как она будет иметь базовый тайтл. ID она имеет. Если мы фетчим весь список публикаций, то пока публикация является драфтом. Как только мы ее переведем из драфта, сможем ее отобразить. Таким образом одним свойством `includePendingChanges` мы уже получаем хорошо работающую фичу.

И также здесь есть оптимизация выделения оперативной памяти для фетч-запроса. Мы можем указать, какие именно поля самих сущностей мы хотим запросить и какие поля сущностей внутри связи нам нужны.

Таким образом через те же механизмы, что были доступны в CoreData, мы можем оптимизировать размер выделяемой оперативной памяти для хранения фетч-запроса и в тот же момент максимально быстро получать доступ к тем полям, которые нам действительно нужны.

Предикаты — мощные compile-time киллер фичи. Они поддерживают подзапросы, джойны и транслируются явно в SQL-запросы. На уровне фетч-дескриптора мы можем настроить такие вещи, как `offset-limit`. Он работает с фолтом и префетчем.

CoreData vs SwiftData	
CoreData	SwiftData
<code>NSManagedObject</code>	<code>@Model</code>
<code>NSPersistentContainer</code>	<code>ModelContainer</code>
<code>NSManagedObjectContext</code>	<code>ModelContext</code>
<code>NSPredicate</code>	<code>#Predicate</code>
<code>NSFetchRequest</code>	<code>FetchDescriptor</code>
<code>NSSortDescriptor</code>	
<code>NSFetchedResultsController</code>	<code>@Query</code>

29

Улучшение синтаксиса предикатов

Интеграция со SwiftConcurrency

Сложно объяснить весь блеск SwiftConcurrency в связке со SwiftData, не объяснив всю нищету того, как работала многопоточная CoreData. Несмотря на то, что я ее очень люблю,

уровень сложности и количество проблем, которые она приносила, — это просто невыносимо.

CoreData работает так: один поток — один контекст. Соблюдай правила — ничего не будет болеть. Непонятные типы хранилища XML, которые идут нам еще с macOS, недоступны в iOS, но все еще есть в списке. Полностью весь движок на Objective-C runtime, который может в любой момент выстрелить — и мы даже не поймем, что произошло. И концептуально с iOS 5 CoreData не менялась.

Последнее время люди делились на тех, кто не понимает CoreData, и теми, кто не понимает Realm. Как только появился Swift с Codable, все такие: будем хранить все в файлах! И кажется, это не совсем то, чего хочет от нас Apple. Давайте посмотрим, как работает многопоточность в CoreData.

```
func update(post: CoreData.Post, message: String) async throws {
    guard let context = post.managedObjectContext else {
        throw CoreData.CoreError.noContext
    }
    try await context.perform {
        post.message = message
        try context.save()
    }
}
```

Многопоточность в CoreData работает предсказуемо. Всегда одному потоку соответствует один контекст и объекты между потоками передаются через Object ID.

Это происходит из-за того, что Manage Object и Manage Context не потокобезопасны. Казалось бы, Apple вам в iOS 5 добавила такие классные штуки, как Context Perform и Parent Context. Но мы не можем делать хорошо CoreData, потому что это требует кучу когнитивной нагрузки и полного понимания того, как работает многопоточность в iOS.

Я часто видел, как разработчики брали конкурентную очередь, ассоциировали ее с контекстом и у них все отлично работало. Они считали, что удовлетворяют правилу «один поток, один контекст»: и у них, и у тестировщиков все отлично работало. А у пользователей все дико крашилось. Потому что разработчики забывали, что в релизной конфигурации под нагрузкой конкурентная очередь выполняется на разных потоках.

Если в параллель начинается работа над одним объектом внутри одного контекста из разных потоков, то мы получаем в лучшем случае неконсистентные данные, а в худшем — краш на сейве. И это отловить было очень сложно. И казалось бы, можно делать вот так, но это будет некрасиво. И все пытаются сделать красиво, но крашатся.

В SwiftData мы получили, что теперь контексты связаны с Queue явно. И через Queue они связаны с очередями. Ушел непонятный тип хранилища XML, и у нас полностью нативная поддержка Swift. Вспоминаем предикаты, макросы, все в Compile Time Safety.

SwiftConcurrency со SwiftData связаны так: контекст SwiftData ассоциирует с Queue. Каждый раз, когда мы создаем контекст, у нас SwiftData запоминает, в какой Queue создан этот контекст, и запоминает это в рамках актора.

Если мы создаем в background-потоке private-контекст, он ассоциируется с private Queue. Если в главном потоке, он ассоциируется с main Queue, которая serial.

SwiftData предоставляет ModelContainer. ModelContainer внутри имеет main-контекст, который атрибутирован main-актором.

С main-актором все знакомы, понимаем, что будет, если мы попробуем main-контекст использовать на background-потоках. По бэкграунду есть отдельный актор — ModelActor. Он содержит протокол, ModelContainer и ModelExecutor.

ModelExecutor — это макрос. Мы можем атрибутировать этим макросом свой актор и посмотреть, что внутри конструктора будет создаваться контекст. Так контекст будет ассоциироваться с Queue и передаваться в default serial ModelExecutor.

Есть подсказка к тому, как работает @ModelActor, — serial. Но кто это такой? Что за default serial ModelExecutor? Он создается только в конструкторе ModelActor, принимает контекст, который мы создали прямо там же рядом, в этой же Queue, и, как говорит нам документация, он безопасно работает с хранилищем.

DefaultSerialModelExecutor работает серийно на нужной очереди. Это именно то же самое, что было в CoreData с методами perform, только теперь это работает очевидно. И SwiftConcurrency будет бить по рукам, если мы будем использовать это некорректно. Наконец-то 🧨

```
@ModelActor
actor MobiusModelActor {
    let modelExecutor: any ModelExecutor
    let modelContainer: ModelContainer
    init(modelContainer: ModelContainer) {
        let modelContext = ModelContext(modelContainer)
        modelExecutor = DefaultSerialModelExecutor(modelContext: modelContext)
        self.modelContainer = modelContainer
    }
}
```

CoreData и SwiftData вместе

CoreData и SwiftData могут жить вместе и писать в один стор. SwiftData может читать из сторов, в которые в этот же момент пишет CoreData. А может работать и наоборот — записанные данные из SwiftData будут доступны в CoreData.

Apple дала классный инструмент, который генерирует из наших Core Data Model SwiftData-классы. Но нужно эти SwiftData-классы положить в другой namespace. Например, в другой модуль или обернуть их в enum.

Я советую их оборачивать в структуру или в enum, потому что при следующей миграции будет очень удобно, что есть несколько enum для каждой версии SwiftData.

И значит, SwiftData автоматически вам сделает миграцию.

Это все замечательно работает, если у вас минимальный таргет iOS 17. Но кажется, что ничего волшебного в SwiftData нет, чего не было в CoreData.

В своем проекте я измерил производительность — и оказалось, что ни прироста, ни деградации производительности нет. Имеет ли смысл вообще поднимать таргет до iOS 17 или продавать идею миграции на SwiftData? Есть ли пофит, кроме безопасности?

Как вы думаете, насколько SwiftData лучше CoreData? Ноль. В тех местах, где утыкается CoreData, — там же утыкается SwiftData. Все идентично до нюансов.

Детали SwiftData

Изначально я мерил performance на CoreData-проекте и увидел, что performance идентичный. Записал полностью логи SQL-запросов и мигрировал этот проект полностью на SwiftData. Удалил CoreData для честности эксперимента: может, они на интеропе тормозят. И прогнал то же самое.

Записываем логи CoreData примерно так:

```
@Model
final class MobiusEntity1 {
    var timestamp: Date
    var oneToOne: MobiusEntity2
    var oneToMany: [MobiusEntity3]
    init(timestamp: Date) {
        self.timestamp = timestamp
        self.oneToOne = MobiusEntity2(timestamp: timestamp)
    }
}
```

```
        self.oneToMany = [MobiusEntity3(timestamp: timestamp)]
    }
}
```

В launch-аргументы пишем `com.apple.coredata.sqldebug`. Ключ у `SwiftData` такой же. И логи — `coredata.sql.begin`, `coredata.sql.insert`. Логи идентичные, файлы идентичные.

Вывод: внутри `SwiftData engine.coredata`. Как, блин, они это сделали? И плохо ли это?

Неделю я был расстроен, потом отошел и начал думать. Внутри у нас `SQLite`. Все проблемы `CoreData` всегда были именно в проблемах `SQLite`, потому что при любых сложностях и работе с диском мы упираемся в его скорость.

`SQLite` — классный фреймворк и самая популярная встраиваемая база данных. `SQLite` на 100% покрыт тестами — это общеизвестный факт. Каждый релиз прогоняется 5 миллионов тестов — все виды тестирования. С 2009 года `SQLite` полностью покрыт тестами — это написанный на C оптимизированный код, который можно читать. Единственный код на C, который можно читать, из тех, что я встречал. И кроме того, этот код в публич-домене. То есть он доступен, мы его можем читать, мы его можем использовать: открытая лицензия.

Но вопрос: как они это сделали? Если есть `SwiftData`, которая использует актуальные функции языка и под капотом `CoreData`, давайте сделаем свою `SwiftData` с блэкджеком и поддержкой iOS 14. У нас везде макросы.

Хочется понять, что происходит под капотом. Раскрываем макрос:

```
@Model
final class Post {

    @Attribute(.unique)
    var id: String
    var message: String
    var timestamp: Date

    @Transient
    var _$backingData: any BackingData<Post> = Post.createBackingData()
}
```

Видим: `BackingData`, `PostCreateBackingData`. Что такое `BackingData`? Как нам говорит открытая документация, это какой-то внутренний, ненужный класс. Там есть приватный класс, который реализует протокол `BackingData`, `DefaultBackingData`.

Если мы знаем класс, то знаем Objective-C, идем в рефлексию, находим очень странное property:

```
public extension BackingData {  
    var managedObject: NSManagedObject? {  
        guard let object = getMirrorChildValue(  
            of: self,  
            childName: "_managedObject") as? NSManagedObject else {  
            return nil  
        }  
        return object  
    }  
}
```

Если мы его приведем к `ManagedObject`, то получаем, что каждый раз, когда создается `SwiftData`-класс, под капотом создается `CoreData`-сущность без контекста. Но конструктор специфический, я его не сразу узнал, потому что раньше `Entity` приходилось строкой писать.

За счет конструктора мы можем в `SwiftData` оторвать модели от контекста, потому что так же, как было с `Alloc` и `Nid`, мы отдельно создаем модельку и отдельно `insert`-им ее в контекст.

Мы можем убедиться в том, что у созданного объекта есть ассоциированный `managed-object`. То есть мы создали `SwiftData`-объект `POST` и у него же можем за счет нашей `extension` получить `CoreData`-объект.

Мы поняли то, что `CoreData` — это всего лишь текущая реализация в `SwiftData` и `BackingData` позволяет Apple подменить ее другими реализациями. То есть если сейчас `BackingData` имеет только `DefaultBackingData`-класс, то потом может быть `NoSQLBackingData` и это будет иметь свои преимущества в других use-кейсах.

За счет макросов мы поняли, как Apple смогла добиться независимости модели от контекста несмотря на движок `CoreData` под капотом, но это не дало ответ, как реализовать то же самое самим с таргетом ниже `iOS 17`.

Мы все еще не можем сделать свою `SwiftData` на основе `CoreData`, потому что акторы — те, что нам предоставляет `SwiftData`, — доступны в конкретной реализации `iOS 17`, потому что весь обсервинг работает через новый обсерв, который тоже доступен только в `iOS 17`. И, видимо, именно из-за этих концептуальных ограничений и полной совместимости с актуальными `SwiftUI` они так и сделали.

Мы можем разобраться с концептуальными киллер-фичами SwiftData и принести их в CoreData для упрощенной миграции в будущем. Во-первых, хочется получить Code-First-подход. Во-вторых, хочется использовать макросы для Compile-Time-Safety. Хочется отвязать ManagedObjects от ManagedObjects-контекстов, чтобы иметь чистый CRUD и дальше SwiftData была идентична при миграции. И хочется использовать наконец-таки SwiftConcurrency для потока безопасности в CoreData.

И тут мне на ум пришел open source framework CoreStore. Он уже умеет Code-First-подход и выглядит вот так:

```
final class Post: CoreStoreObject {
    @Field.Stored("latitude")
    var id: String
    @Field.Stored("latitude")
    var message: String

    @Field.Relationship("attachments")
    var attachments: [Attachment]
    @Field.Relationship("author")
    var author: User
}
```

После того этот фреймворк нам предоставляет DataStack. То есть Model у нас имел Store и мы описывали списком все модели. Здесь мы не сможем оптимизировать до одной и опустить связи, но по концепции это идентично. Мы просто перечисляем все сущности, и через генерики Compile-Time-Safety достигается по фетчингу.

Пока я не видел ни одной реализации и не смог сам быстро реализовать похожие предикаты Compile-Time-Safe, но именно CoreStore уже содержит через Key-Value и через свой DSL безопасный, понятный фетчинг, где у нас есть var через Key-Value. Он понимает контекст, то есть из Favorite мы сравниваем с Bool, а PublishDate мы сравниваем с Date. И это будет валидно, и если мы попробуем сравнить не те типы данных, оно нам скажет, что что-то идет не так.

В CoreStore-библиотеке сортировка работает через Key-Value, предсказуемо и будет валидироваться в Compile-Time. Базовое Compile-Time мы получаем в Safety, и кажется, что, если пойти чуть глубже, то можно попробовать реализовать и Macro-Predicator на основе уже готового API для этого DSL. Но я до этого пока не дошел.

Самое важное, что хочется получить в CoreData, что есть в SwiftData, — это потокобезопасность. Кажется, что за счет того, что у нас есть понимание, как сделаны акторы в SwiftData, мы можем принести ту же логику в CoreData. Нам ничего не мешает

сделать Serial Executor на Serial Queue, который будет просто вызывать контекст perform, а он уже работает через Serial Execution, и это все обернуть в актор.

Если мы не сможем использовать по-другому CoreData, если мы заставим всех использовать ее через акторы, то мы получим потокобезопасность в CoreData. То же самое, что мы имеем в SwiftData. А выстрелить в ногу с CoreData можно будет всегда. И в целом, можно сказать, что мы пришли к цели.

В итоге мы получили:

1. Code First-подход — самое простое и базовое, что нам поможет потом в миграции.
2. Макросы и Compile Time Safety.
3. Интеграцию SwiftConcurrency. Да, не в том формате, в котором было это в SwiftData. Но если соблюдать те же самые правила и те же самые подходы, то, во-первых, мы сможем обучить наших коллег использовать те концепции, которые к нам придут со SwiftData. А во-вторых, мы сможем ограничивать и снижать когнитивную нагрузку при работе с многопоточной.

Подводя итоги

SwiftData меня разочаровала только при первом приближении, а потом я понял, что она действительно стоит того. Это не новый фреймворк, это та CoreData, которую мы ждали с iOS 5. Это правильная CoreData с использованием Swift, современных подходов к проектированию, программированию и всех самых главных возможностей.

За счет того, что Apple дала нам тулинг, мы можем достаточно просто мигрировать существующие проекты с громадными .xcdatamodel-ами на SwiftData.

В SwiftData реализовано все лучше, включая миграции. И за счет того, что мы имеем под капотом CoreData, а еще под капотом SwiftData, это работает предсказуемо. И мы будем уверены в том, что сможем найти решение, потому что мы сможем включить SQL debug и посмотреть, что за запросы идут.

Каждый раз, когда мы утыкались в производительность CoreData, помогал именно SQL debug. Мы просто смотрели логи и пытались оптимизировать запросы.

За счет того, что внутри SQLite есть тулинг для просмотра красивых схем вашей базы данных, для автоматического анализа, для профилирования ваших select-запросов. Есть даже инструменты, правда платные, которые самостоятельно могут профилировать все ваши запросы и давать советы по оптимизации. И это очень классно, что под капотом в

целом у нас остается надежный проверенный инструмент, который оптимизирован донельзя. Это С. Но тем не менее в SwiftData появились классные фичи, которые в CoreData не получится встроить.

Появились автосохранение, миграции, и SwiftData работает через макросы. Теперь у нас есть concurency и есть макросы, которые нам позволяют работать.

Заходите в комментарии, если есть вопросы или истории, которыми хочется поделиться!

Теги: coredata, swiftdata, swift, swiftui, ios

Хабы: [Блог компании TINKOFF](#), [Разработка под iOS](#), [Разработка мобильных приложений](#), [Swift](#), [Разработка под macOS](#)

Редакторский дайджест

Присылаем лучшие статьи раз в месяц

Электронпочта



TINKOFF

IT's Tinkoff — просто о сложном



16

12

Карма

Рейтинг

Андрей @Azon
Автор iOS Broadcast, Разработчик в Тинькофф

 Комментарии 8

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ



finindie

16 часов назад

Будет ли пенсия у родившихся в восьмидесятых?

 Простой  12 мин  28K

 +148

 88

 397



yard

22 часа назад

Неужели Banki.ru сливают ваши данные спамерам? Или как не угодить в ловушку микрозаймов

 Средний  6 мин  13K

Кейс

 +113

 49

 151



GeeksCat

21 час назад

Blade Runner 2049 — это экранизация Набокова

 Простой  8 мин  6.8K

Мнение

 +55

 34

 33



fisher

6 часов назад

freenginx: комментарии от Макса Дунина

 Простой  4 мин  5.3K

Интервью

 +44

 11

 26



DAN_SEA

21 час назад

Range Extender на NRF24L01+PA+LNA: обмен текстовыми сообщениями между устройствами там, где нет сотовой связи

 Простой  11 мин  3.6K

Обзор

 +39

 51

 26



fellow_pablo
22 часа назад

Эргономика рабочего места инди-разработчика, или как я избавился от боли в спине



Простой



5 мин



5.4K

Кейс



+28



40



37



BabayMazay
1 час назад

На пути к самодельным радиолампам. Стеклодувные операции



Средний



7 мин



647

Тutorial



+18



10



0



un1t
23 часа назад

Как я выучил словарь на 9000 слов



7 мин



9.5K



+17



59



30



olya_duel
22 часа назад

Совершенствуем UX. Разговоры о серьёзном с администраторами



8 мин



774



+16



16



0



SLY_G
23 часа назад

Уэбб может напрямую проверить одну из теорий о тёмной материи



5 мин



2.3K

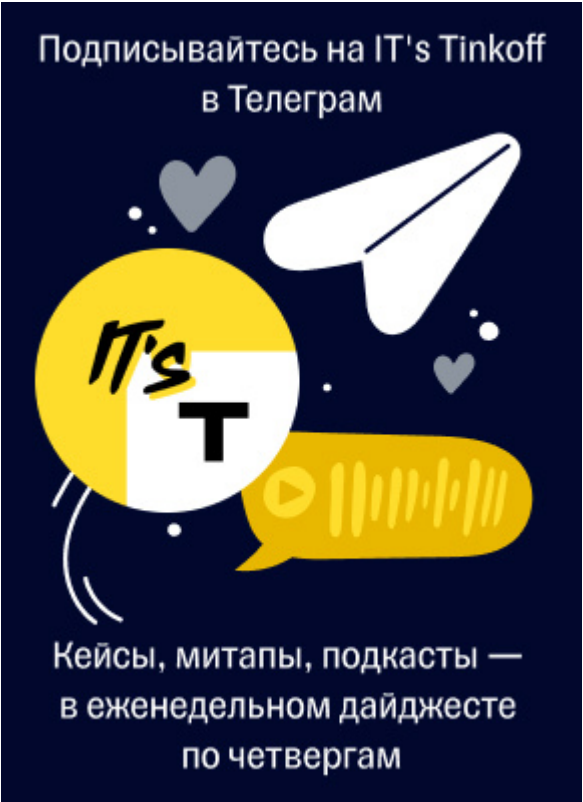
Перевод

Показать еще

ИНФОРМАЦИЯ

Сайт	www.tinkoff.ru
Дата регистрации	26 октября 2011
Дата основания	18 ноября 2005
Численность	свыше 10 000 человек
Местоположение	Россия


ВИДЖЕТ




БЛОГ НА ХАБРЕ

23 часа назад

Как выжать максимум из Confluence. Глава вторая

 2.6K

 6

13 фев в 17:53

Опровергаю пять архитектурных заблуждений

 11K

 15

8 фев в 16:33

Java Digest #9

 3.2K

 1

6 фев в 18:00

Кто ты, SwiftData

 2.5K

 8

1 фев в 14:46

Scala Digest. Выпуск 13

 1.1K

 2

Ваш аккаунт

Войти
Регистрация

Разделы

Статьи
Новости
Хабы
Компании
Авторы
Песочница

Информация

Устройство сайта
Для авторов
Для компаний
Документы
Соглашение
Конфиденциальность

Услуги

Корпоративный блог
Медийная реклама
Нативные проекты
Образовательные
программы
Стартапам



Настройка языка

Техническая поддержка

