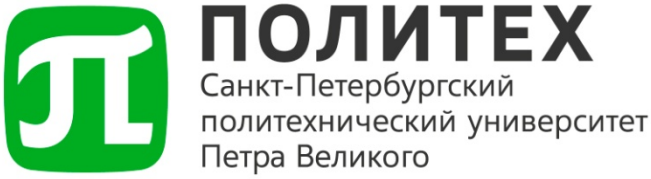
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
ВШ программной инженерии



КУРСОВАЯ РАБОТА

«Безопасное хранение персональных данных на устройствах IOS»

по дисциплине «Защита информации»

Выполнил
Студент 3530202/90202 группы А.М. Потапова

Руководитель Б.М. Медведев

Санкт-Петербург
2023 г.

ЗАДАНИЕ
НА ВЫПОЛНЕНИЕ КУРСОВОЙ РАБОТЫ
по дисциплине
«Защита информации»

студенту группы 3530202/90202 Потапова Алина Михайловна
(номер группы) *(фамилия, имя, отчество)*

1. Тема работы Безопасное хранение персональных данных на устройствах с операционной системой IOS

2. Срок сдачи студентом законченной работы *14.04.2023*

3. Исходные данные к работе

4.Содержание пояснительной записки (перечень подлежащих разработке вопросов):
1) Обзор литературы по выбранному направлению, содержащий описание угроз безопасности, рисков, связанных с реализацией угроз, анализ основных решений для защиты информации. 2) Цель работы и решаемые задачи. 3) Алгоритмы защиты и сценарии использования. 4) Программные средства защиты информации. 6) Результаты работы 7) Выводы.

Примерный объем пояснительной записки 15 страниц машинописного текста

5. Перечень графического материала (с указанием обязательных чертежей и плакатов) *не предоставляется*

6. Консультанты

7. Дата получения задания: «17» февраля 2023 г.

Руководитель _____ *Б.М. Медведев*
(подпись) *(инициалы, фамилия)*

Задание принял к исполнению _____ *А. М. Потапова*
(подпись) *(инициалы, фамилия)*

17.02.2023
(дата)

СОДЕРЖАНИЕ

Обзор литературы

Каждый год значимость безопасности хранения пользовательских данных в мобильных приложениях возрастает. С ростом осведомленности пользователей об угрозах утечек конфиденциальной информации разработчики приложений должны обеспечивать безопасность хранения данных, используя различные методы защиты.

Современные устройства IOS [1] содержат огромное количество конфиденциальной информации, такой как контакты, сообщения, фотографии, видео, финансовые данные, личные записи и многое другое. К сожалению, это также делает их целью для злоумышленников и киберпреступников, которые могут использовать эту информацию для кражи личных данных, финансовых мошенничеств, взлома устройства и других угроз.

Несмотря на то, что IOS считается одной из самых безопасных платформ для мобильных устройств, она не является абсолютно защищенной от всех видов угроз безопасности. Некоторые из уязвимостей могут быть вызваны некорректной настройкой устройства, ошибками в приложениях, несанкционированным доступом к устройству и другими причинами. Таким образом, приоритетной задачей разработчиков приложений становится обеспечение безопасности хранимых данных.

Виды персональных данных

Для начала, рассмотрим основные виды персональных данных пользователей iOS девайсов:

- Контактная информация: номера телефонов, адреса электронной почты, адреса проживания и рабочие адреса.
- Финансовые данные: информация о банковских счетах, кредитных картах и других финансовых средствах.

- Информация об авторизации: учетные записи, пароли, ключи и сертификаты.
- Личные фотографии и видео.
- Данные о местоположении устройства в определенный момент времени.
- История браузера, посещенные сайты и другая информация о сетевой активности устройства.
- Данные о здоровье и фитнесе, включая данные о сердцебиении, соне, физической активности и др.

Кража таких данных может привести к серьезным последствиям, таким как потеря денежных средств, утечка конфиденциальной информации и т.п.

Оценка рисков реализации угроз

Рассмотрим наиболее распространенные методы кражи перечисленных выше пользовательских данных с оценкой уровня опасности:

- Утеря или кража устройства – *высокий уровень опасности*. Если устройство IOS потеряно или украдено, злоумышленник сможет без особых усилий получить доступ к конфиденциальной информации владельца мобильного устройства.
- Вредоносное ПО – *высокий уровень опасности*. Вредоносное ПО на устройствах IOS может быть установлено через приложения из App Store, веб-сайты или электронную почту. Вредоносное ПО может использоваться для сбора конфиденциальной информации, такой как логины и пароли, и для управления устройством без ведома пользователя.
- Фишинг – *средний уровень опасности*. Фишинг – это мошенническая попытка получить личную информацию, такую как логины и пароли, путем подделки электронных писем, текстовых сообщений и веб-страниц. Несмотря на то, что устройства IOS имеют высокий уровень защиты от

фишинга, пользователи все еще могут быть подвержены этой угрозе, если они не будут осторожны при открытии электронных сообщений или переходе на подозрительные веб-сайты.

- Несанкционированный доступ к данным приложений – *средний уровень опасности*. Если приложение не было разработано с учетом безопасности данных, злоумышленник может получить доступ к конфиденциальной информации, которая хранится в приложении.
- Сетевые атаки – *низкий уровень опасности*. Сетевые атаки могут использоваться для получения доступа к конфиденциальной информации, которая передается между устройством и сервером. Однако, с учетом высокого уровня защиты в IOS, эта угроза не является частой, и пользователи обычно не должны беспокоиться об этом.

Способы устранения угроз

Для устранения перечисленных рисков пользователю можно применять следующие способы:

- Использование паролей и Touch/Face ID для защиты устройства. Это поможет предотвратить несанкционированный доступ к устройству и сохранить персональные данные.
- Использование защищенных соединений. Приложения должны использовать защищенные протоколы, такие как HTTPS, для передачи данных между устройством и сервером.
- Ограничение доступа к персональным данным. Приложения должны иметь минимальный доступ к персональным данным и запросить разрешение у пользователя на доступ к ним.
- Регулярное обновление операционной системы и приложений. Это поможет обеспечить устранение уязвимостей и исправление ошибок в безопасности данных.

- Резервное копирование данных. Пользователи должны регулярно резервировать свои данные, чтобы в случае потери или кражи устройства сохранить доступ к своим персональным данным.
- Тщательный выбор приложений. Пользователи должны скачивать приложения только из официальных источников, чтобы избежать установки вредоносных программ.
- Регулярное удаление ненужных данных. Пользователи должны удалять ненужные данные, такие как старые сообщения или фотографии, чтобы уменьшить риск кражи и потери данных в случае утери или кражи устройства.
- Использование VPN для защиты соединения в общественных сетях Wi-Fi. VPN шифрует соединение и помогает обезопасить передачу данных в общественных сетях Wi-Fi.

Разработчику, в свою очередь, приходится отталкиваться от формата разрабатываемого ПО и используемых технологий. Но, для общих случаев предлагаются следующие решения:

- Хранение паролей и других конфиденциальных данных в безопасном виде с использованием шифрования данных.
- Использование защищенного соединения для передачи данных между приложением и сервером.
- Использование технологии обратимого шифрования для защиты данных в случае утечки устройства.
- Ограничение доступа к базе данных только авторизованным пользователям и разрешение на доступ только необходимой информации.
- Использование механизмов аутентификации для защиты данных от несанкционированного доступа.
- Ограничение доступа к приложению только через защищенный пароль или Touch ID/Face ID.
- Хранение критических данных на сервере в зашифрованном виде.

- Ограничение возможности копирования и экспорта данных из приложения.
- Регулярное обновление приложения для устранения обнаруженных уязвимостей и повышения уровня безопасности.
- Проведение аудита безопасности приложения с целью выявления возможных уязвимостей и проблем в защите данных.

В моем случае приложение использует следующие пользовательские данные: фото/видео контент и текстовые записи. В качестве паттерна проектирования используется MVVM [4]. А в качестве базы данных – фреймворк CoreData [5]. Сторонние сервисы, предоставляющие серверы для хранения данных пользователей было решено не привлекать, с целью устранения рисков кражи и минимизации финансовых затрат на разработку приложения.

Таким образом, защита данных в моем ПО сводится к следующим методам:

- Шифрование данных. Все пользовательские данные, такие как фото/видео и текстовые записи, зашифрованы, используя алгоритм шифрования AES. Это позволит обезопасить данные в случае несанкционированного доступа к устройству.
- Резервное копирование данных. Резервное копирование пользовательских данных на облачный сервис iCloud [7] позволит сохранить копию данных в случае потери устройства или повреждения файлов. iCloud использует сильное шифрование для защиты данных в пути и в покое. Данные хранятся в зашифрованном виде на серверах Apple, а доступ к ним предоставляется только после прохождения проверки подлинности.
- Ограничение доступа к данным. Ограничение доступа к пользовательским данным можно осуществить через различные механизмы, такие как пароли, PIN-коды, Touch ID и Face ID. Несанкционированный

доступ к данным можно также предотвратить путем ограничения доступа к файловой системе iOS и использования системного шифрования данных.

- Обновление приложения. Важно периодически обновлять приложение, чтобы закрыть обнаруженные уязвимости в безопасности. Обновления могут включать исправления ошибок, а также улучшения безопасности.

В данной работе я рассмотрю перечисленные методы и интегрирую их в свое приложение.

Цели работы и решаемые задачи

Цель: обеспечить надежное хранение пользовательских данных в приложении, предназначенном для записи воспоминаний и хранения фото/видео контента, используя механизмы шифрования данных и резервное копирование.

Задачи:

- Реализовать безопасный функционал сохранения и загрузки данных в локальную базу данных фреймворка CoreData используя паттерн MVVM.
- Разработать механизм шифрования для защиты пользовательских данных.
- Реализовать функционал создания резервных копий данных и их восстановления из облачного хранилища iCloud для предотвращения потери данных в случае сбоя приложения или устройства.
- Протестировать приложение на наличие уязвимостей и ошибок в системе безопасности.
- Получить у пользователя доступ к необходимым персональным данным.
- Реализовать механизм обновления приложения с целью исправления обнаруженных уязвимостей и ошибок в системе безопасности.

Описание разработки

Модель CoreData

В первую очередь создадим модель «ItemMO» (воспоминание) в базе данных, используя фреймворк CoreData.

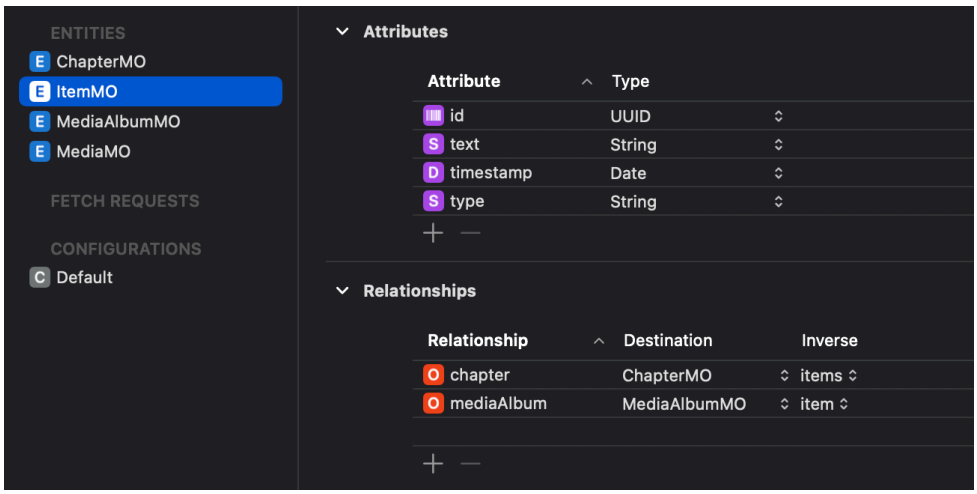


Рисунок 1. Модель «Item» в CoreData.

Объявление созданного класса «ItemMO» на языке Swift [2] выглядит следующим образом:

```
import Foundation
import CoreData

@objc(ItemMO)
public class ItemMO: NSManagedObject {
    @NSManaged public var id: UUID?
    @NSManaged public var timestamp: Date?
    @NSManaged public var text: String?
    @NSManaged public var type: String?
    @NSManaged public var mediaAlbum: MediaAlbumMO?
    @NSManaged public var chapter: ChapterMO?
}
```

Рисунок 2. Объявление класса «ItemMO»

Шифрованию подлежат следующее поля класса:

- text - текст, связанный с элементом.
- mediaAlbum - связь с сущностью MediaAlbumMO, представляющей коллекцию медиафайлов.

Модель представления «ItemViewModel», представленная на *рисунке 3* отвечает за работу с воспоминаниями (items), связанными с конкретной главой (chapter), и обеспечивает методы для загрузки, сохранения и обработки ошибок при сохранении данных.

```
import Foundation
import CoreData

class ItemViewModel: ObservableObject {
    private let viewContext = PersistenceController.shared.viewContext
    @Published var items: [ItemMO] = []
    var chapter: ChapterMO

    public var alert = false
    public var alertMessage = ""

    init(chapter: ChapterMO) {
        self.chapter = chapter
    }

    func fetchItems() {
        items = chapter.itemsArray
    }

    func save() {
        do {
            try viewContext.save()
            alert = false
        } catch {
            alert =  true
            alertMessage = "Saving data error"
        }
    }

    ...
```

Рисунок 3. Фрагмент кода, описывающий модель представления «ItemViewModel»

Ключевое свойство:

- viewContext — это контекст, который используется для сохранения данных. Он создается через PersistenceController, который управляет CoreData в приложении.

Ключевые методы:

- fetchItems() – метод для загрузки всех элементов (items), относящихся к текущей главе (chapter), в свойство items.
- save() – метод для сохранения изменений в CoreData. Если сохранение прошло успешно, то свойство alert устанавливается в false, иначе – в true, а свойство alertMessage присваивается значение "Ошибка сохранения".

Шифрование данных

Добавим методы шифрования и расшифровки текста с использованием алгоритма AES:

```swift
// Функция для генерации ключа для AES-шифрования
func generateAESKey() -> SymmetricKey {
    return SymmetricKey(size: .bits256)
}

// Функция для шифрования текста с помощью AES
func encryptAES(text: String, key: SymmetricKey) throws -> Data {
    let textData = Data(text.utf8)
    let sealedBox = try AES.GCM.seal(textData, using: key)
    return sealedBox.combined!
}

// Функция для расшифровки текста, зашифрованного с помощью AES
func decryptAES(data: Data, key: SymmetricKey) throws -> String {
    let sealedBox = try AES.GCM.SealedBox(combined: data)
    let decryptedData = try AES.GCM.open(sealedBox, using: key)
    return String(data: decryptedData, encoding: .utf8)!
}
```

Рисунок 4. AES-шифрование текста.

Описание методов необходимых для AES-шифрования:

- Метод generateAESKey генерирует случайный ключ для использования в AES-шифровании.
- Метод encryptAES шифрует текст с помощью AES-алгоритма, используя ключ, и возвращает зашифрованные данные в виде объекта Data.
- Метод decryptAES расшифровывает данные с использованием того же ключа, который использовался для шифрования, и возвращает расшифрованный текст в виде строки.

Аналогично поступим с зашифровкой и расшифровкой фото:

```swift
// Генерация ключа AES
func generateAESKey() -> SymmetricKey {
    return SymmetricKey(size: .bits256)
}

// Шифрование фото AES
func encryptPhoto(photo: UIImage, key: SymmetricKey) throws -> Data {
    guard let imageData = photo.jpegData(compressionQuality: 1) else { throw EncryptionError.encodingError }
    let sealedBox = try AES.GCM.seal(imageData, using: key)
    return sealedBox.combined!
}

// Расшифровка фото AES
func decryptPhoto(data: Data, key: SymmetricKey) throws -> UIImage {
    let sealedBox = try AES.GCM.SealedBox(combined: data)
    let decryptedData = try AES.GCM.open(sealedBox, using: key)
    guard let image = UIImage(data: decryptedData) else { throw EncryptionError.decodingError }
    return image
}
```

Рисунок 5. AES-шифрование фото.

Резервное копирование данных

Для начала создадим класс, соответствующий записи в CloudKit:

```
import CloudKit

class ItemRecord: CKRecord {
    static let recordType = "Item"

    init(item: ItemMO) {
        super.init(recordType: ItemRecord.recordType)
        self.setValue(item.id, forKey: "id")
        self.setValue(item.timestamp, forKey: "timestamp")
        self.setValue(item.text, forKey: "text")
        self.setValue(item.type, forKey: "type")

        // переносим информацию об альбоме медиафайлов
        if let mediaAlbum = item.mediaAlbum {
            let mediaAlbumRecord = MediaAlbumRecord(mediaAlbum: mediaAlbum)
            self.setValue(mediaAlbumRecord, forKey: "mediaAlbum")
        }
    }

    required init(fromRecord record: CKRecord) {
        super.init(recordType: ItemRecord.recordType)
        self.setValue(record.value(forKey: "id"), forKey: "id")
        self.setValue(record.value(forKey: "timestamp"), forKey: "timestamp")
        self.setValue(record.value(forKey: "text"), forKey: "text")
        self.setValue(record.value(forKey: "type"), forKey: "type")

        // переносим информацию об альбоме медиафайлов
        if let mediaAlbumRecord = record.value(forKey: "mediaAlbum") as? CKRecord {
            let mediaAlbum = MediaAlbumMO(mediaAlbumRecord: mediaAlbumRecord)
            self.setValue(mediaAlbum, forKey: "mediaAlbum")
        }
    }
}
```

Рисунок 6. Класс, соответствующий «ItemMO» в CloudKit.

В классе «ItemMO» создадим функцию, которая создает запись в CloudKit на основе данных текущего объекта «ItemMO»:

```
import CloudKit

extension ItemMO {
    func createItemRecord() -> ItemRecord {
        return ItemRecord(item: self)
    }
}
```

Рисунок 7. Создание записи в CloudKit.

В метод сохранения объекта в локальную базу данных, который представлен на *рисунке 3*, добавим функционал по переносу данных в iCloud. Сначала вызываем метод save() для сохранения изменений в локальной базе данных. Затем создаем запись объекта в формате ItemRecord и сохраняем ее в частной базе данных CloudKit через объект privateCloudDatabase.

Метод save() принимает созданный объект записи и обработчик завершения, который вызывается после сохранения записи в iCloud. Если возникает ошибка при сохранении в iCloud, обработчик выводит сообщение об ошибке, иначе выводится сообщение об успешном сохранении. Реализация представлена на *рисунке 8*:

```
func save() {
    do {
        try viewContext.save()
        alert = false

        // переносим данные в iCloud
        let itemRecord = self.createItemRecord()
        let cloudDatabase = CKContainer.default().privateCloudDatabase

        cloudDatabase.save(itemRecord) { (record, error) in
            if let error = error {
                // обработка ошибок
                print("Error saving record to iCloud: \(error.localizedDescription)")
            } else {
                print("Record saved to iCloud: \(record?.recordID.recordName ?? "unknown")")
            }
        }

    } catch {
        alert =  true
        alertMessage = "Saving data error"
    }
}
```

Рисунок 8. Сохранение записи в CloudKit.

Получение доступа к данным пользователя

Для получения доступа к библиотеке фотографий пользователя, добавим поле ‘Privacy – Photo Library Usage Description’ в свойства нашего проекта:

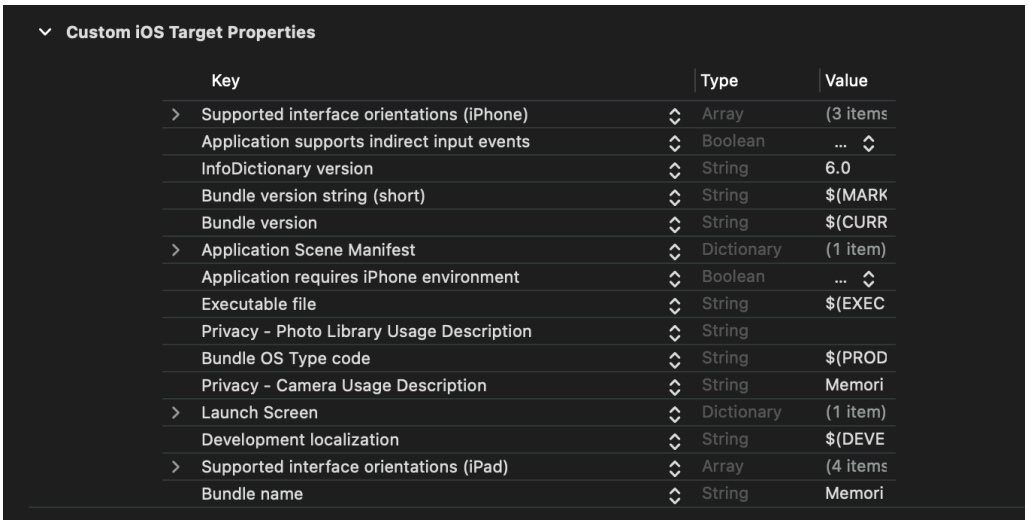


Рисунок 9. Перечень свойств проекта.

В контроллер представления ответственный за выбор фотографий добавим метод setup(), содержащий запрос разрешения на доступ к фото библиотеке пользователя.

```
func setup() {
    PHPhotoLibrary.requestAuthorization(for: .readWrite) { [self] (status) in
        DispatchQueue.main.async { [self] in
            switch status {
            case .denied:
                libraryStatus = .denied
            case .authorized:
                libraryStatus = .approved
            case .limited:
                libraryStatus = .limited
            default: libraryStatus = .denied
            }
        }
    }
}
```

Рисунок 10. Код запроса разрешения на доступ к фото библиотеке.

Функция ‘requestAuthorization’ запрашивает доступ к библиотеке и вызывает блок кода, когда пользователь принимает или отклоняет запрос доступа. В данном случае, при вызове блока кода, свойство libraryStatus устанавливается в одно из трех возможных значений в зависимости от ответа пользователя: ‘approved’ – разрешение получено, ‘denied’ – доступ запрещен, ‘limited’ – доступ разрешен, но с ограничениями.

Тестирование

Для начала создадим метод, который позволит провести тестирование модели представления в изолированной среде, не зависящей от настоящей базы данных и серверов CloudKit.

```
override func setUpWithError() throws {
    // создание mock-объектов
    mockViewContext = NSManagedObjectContext(concurrencyType: .mainQueueConcurrencyType)
    mockCloudDatabase = MockCloudDatabase()

    // инициализация модели представления
    viewModel = ItemViewModel(chapter: ChapterMO(context: mockViewContext))
    viewModel.viewContext = mockViewContext
    viewModel.cloudDatabase = mockCloudDatabase
}
```

Рисунок 11. Метод для тестирования в изолированной среде.

Перейдем к тестированию функциональности сохранения объекта ItemMO в локальной базе данных CoreData и в облачном хранилище iCloud (*рисунок 12*). В методе создается экземпляр ItemMO, заполняется его поле text, затем метод save() модели представления вызывается для сохранения этого объекта. После вызова метода save(), тест проверяет, что данные были сохранены в локальной базе данных CoreData и в облачном хранилище iCloud. Если количество обновленных объектов в mockViewContext равно 1 и количество сохраненных записей в mockCloudDatabase также равно 1, то тест считается успешным.

```
func testSave_Success() throws {
    // создание объекта ItemMO для сохранения
    let item = ItemMO(context: mockViewContext)
    item.text = "test text"
    viewModel.items.append(item)

    // вызов метода сохранения
    viewModel.save()

    // проверка, что данные были сохранены в CoreData
    XCTAssertEqual(mockViewContext.updatedObjects.count, 1)

    // проверка, что данные были сохранены в iCloud
    XCTAssertEqual(mockCloudDatabase.savedRecords.count, 1)
}
```

Рисунок 12. Тестирование успешного сценария.

Протестируем сценарий, когда происходит ошибка при сохранении данных в CoreData:

```
func testSave_Error() throws {
    // создание объекта ItemMO для сохранения
    let item = ItemMO(context: mockViewContext)
    item.text = "test text"
    viewModel.items.append(item)

    // установка флага, чтобы имитировать ошибку при сохранении в CoreData
    mockViewContext.shouldThrowError = true

    // вызов метода сохранения
    viewModel.save()

    // проверка, что флаг ошибки установлен
    XCTAssertTrue(viewModel.alert)
    XCTAssertEqual(viewModel.alertMessage, "Saving data error")

    // проверка, что данные не были сохранены в iCloud
    XCTAssertEqual(mockCloudDatabase.savedRecords.count, 0)
}
```

Рисунок 13. Тестирование ошибочного сценария.

Также протестируем сохранение объекта ItemMO с media-данными в локальной базе данных CoreData и в iCloud.

```swift
func testSave_WithMedia_Success() throws {
    // Создание объекта ItemMO для сохранения
    let item = ItemMO(context: mockViewContext)
    item.text = "test text"
    item.type = "image"

    // Создание media-данных
    let image = UIImage(systemName: "photo")!
    let imageData = image.jpegData(compressionQuality: 0.8)!

    // Создание media-альбома
    let mediaAlbum = MediaAlbumMO(context: mockViewContext)
    let mediaItem = MediaItemMO(context: mockViewContext)
    mediaItem.data = imageData
    mediaAlbum.addToMediaItems(mediaItem)

    item.mediaAlbum = mediaAlbum

    viewModel.items.append(item)

    // Вызов метода сохранения
    viewModel.save()

    // Проверка, что данные были сохранены в CoreData
    XCTAssertEqual(mockViewContext.updatedObjects.count, 2)

    // Проверка, что данные были сохранены в iCloud
    XCTAssertEqual(mockCloudDatabase.savedRecords.count, 1)

    // Проверка, что media-данные были сохранены в iCloud
    let savedRecord = mockCloudDatabase.savedRecords.first!
    let asset = savedRecord["media"] as! CKAsset
    let savedImageData = try! Data(contentsOf: asset.fileURL!)
    XCTAssertTrue(savedImageData == imageData)
}
```

Рисунок 14. Тестирование успешного сценария с сохранением медиа-данных.

Из результатов симуляции видим, что все тесты прошли успешно.

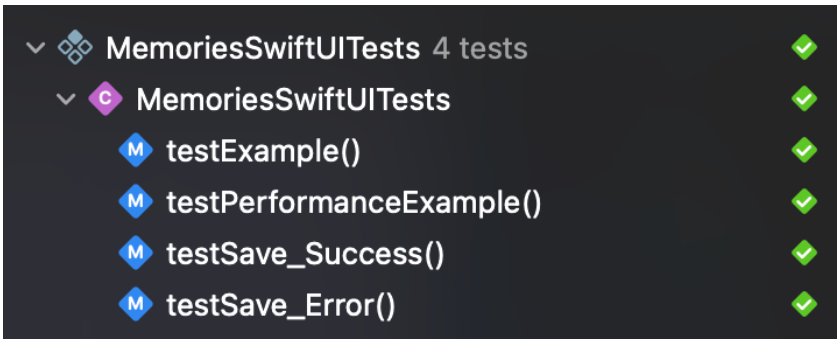


Рисунок 15. Результаты тестирования

Описание программных средств

Данное программное средство разработано на языке Swift и предназначено для операционной системы iOS. Разработка велась в среде разработки Xcode.

Для создания пользовательского интерфейса использован фреймворк SwiftUI [3], который позволяет быстро и удобно создавать интерфейсы для мобильных приложений.

Для шифрования данных использован фреймворк CryptoKit [6], который предоставляет инструменты для криптографических операций, таких как хеширование и шифрование.

Для резервного копирования и синхронизации пользовательских данных использовано облачное хранилище iCloud.

Для управления и хранения данных использован фреймворк CoreData, который обеспечивает доступ к объектно-ориентированной базе данных и предоставляет возможности для автоматического сохранения данных и обработки ошибок.

Для тестирования модели был использован фреймворк XCTest, который предоставляет инструменты для написания и запуска тестов в Xcode. Тестирование включает в себя проверку корректности работы модели, включая сохранение данных в CoreData и iCloud, а также проверку работоспособности функций шифрования и дешифрования данных с помощью CryptoKit.

Результаты

В результате проделанной работы все функциональности были успешно реализованы. Было проведено тестирование при помощи фреймворка XCTest, что помогло обнаружить и исправить ошибки. Качество решения оценивается как высокое, так как были использованы современные технологии и инструменты разработки для iOS, такие как SwiftUI, CryptoKit, CoreData, что позволило создать эффективное и надежное приложение.

Практическая польза данного решения заключается в том, что приложение обеспечивает безопасное хранение и передачу конфиденциальной информации пользователей. Благодаря применению криптографических алгоритмов и защите данных на уровне операционной системы, взлом приложения или утечка информации практически исключены. Ниже представлены скриншоты разработанного приложения:

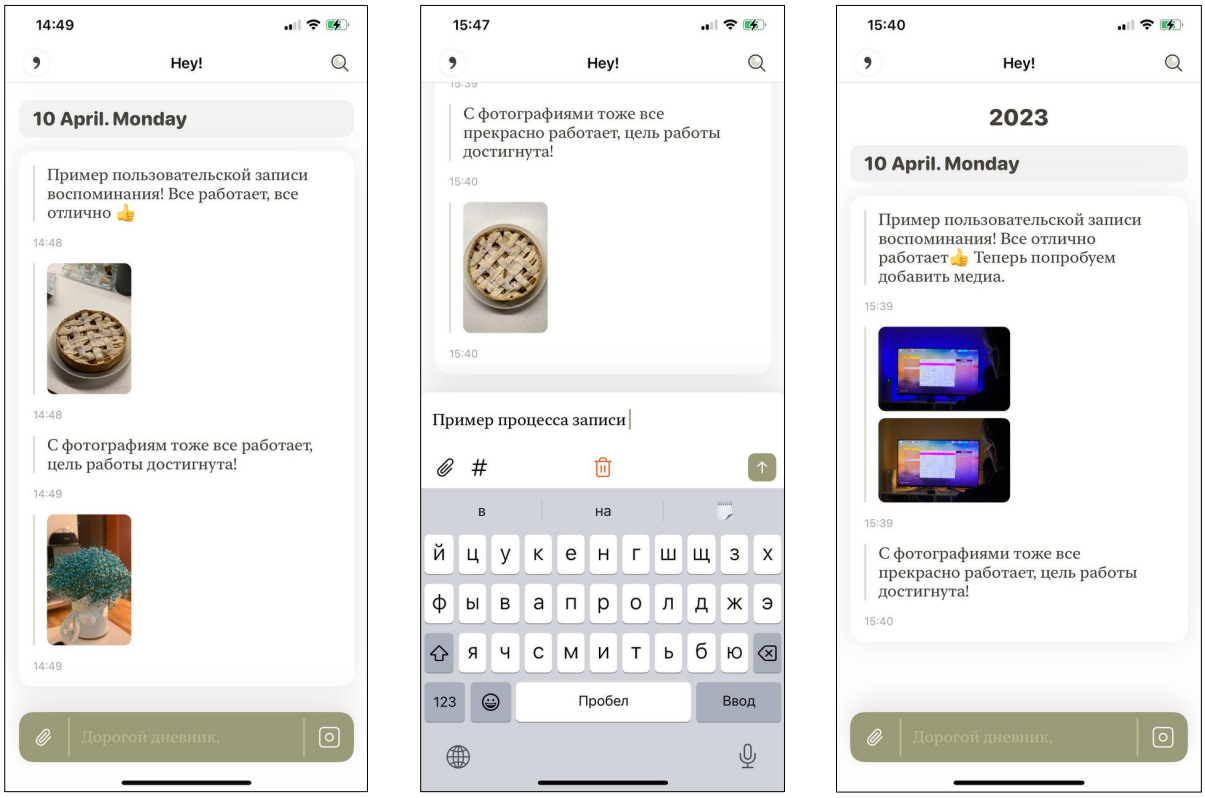


Рисунок 16. Скриншоты разработанного приложения.

Список источников

1. iOS 16 – Apple [Электронный ресурс]. URL: https://www.apple.com/ios/ios-16/
2. Apple Incorporated. About Swift // The Swift Programming Language (Swift 5.7 Edition), 2022. – С. 2-4.
3. SwiftUI Overview - Xcode - Apple Developer [Электронный ресурс]. URL: https://developer.apple.com/xcode/swiftui/ (дата обращения: 12.03.2023)
4. MVVM in iOS Swift – Medium [Электронный ресурс]. URL: https://medium.com/@abhilash.mathur1891/mvvm-in-ios-swift-aa1448a66fb4 (дата обращения: 11.03.2023)
5. Core Data | Apple Developer Documentation [Электронный ресурс]. URL: https://developer.apple.com/documentation/coredata (дата обращения: 14.03.2023)
6. CryptoKit - Apple Developer Documentation [Электронный ресурс]. URL: https://developer.apple.com/documentation/cryptokit (дата обращения: 11.03.2023)
7. iCloud Overview - Apple Developer Documentation [Электронный ресурс]. URL: https://developer.apple.com/icloud (дата обращения: 11.03.2023)