Туториал: Создание бесконечного скролла, работа с пагинацией и динамическим поиском на SwiftUI

Описание

Этот туториал поможет вам шаг за шагом создать галерею изображений с бесконечной прокруткой, кэшированием и динамическим поиском. В данном туториале будет использоваться Unsplash API.

Цель

Научимся собирать экран галереи изображений с возможностью:

- бесконечной прокрутки
- динамического поиска
- кэширования результатов
- и быстрой подгрузки картинок

Проект построен на **SwiftUI** и использует паттерн **MVVM**.

1. UI: View и отображение изображений

Главный экран реализован в `ImageGalleryView`. Он отвечает за отображение строки поиска, сетки изображений и запуск подгрузки при прокрутке.

```
struct ImageGalleryView: View {
    @StateObject private var viewModel = ImageGalleryViewModel()
    ...
```

Основные компоненты:

- **TextField** — строка ввода, которая запускает поиск по нажатию Enter:

```
TextField("Поиск...", text: $viewModel.query, onCommit: {
    viewModel.search(reset: true)
})
```

- **LazyVGrid внутри ScrollView** — отображает изображения в виде сетки из двух колонок:

```
LazyVGrid(columns: [GridItem(.flexible()), GridItem(.flexible())]) {
    ForEach(Array(viewModel.images.enumerated()), id: \.0) { index, image in
        RemoteImageView(urlString: image.urls.small)
    }
}
```

- **onAppear в LazyVGrid** — при появлении определённого элемента вызывается `viewModel.search()` для загрузки следующей страницы. Это и реализует бесконечный скролл:

```
let thresholdIndex = viewModel.images.index(
    viewModel.images.endIndex,
    offsetBy: -6,
    limitedBy: viewModel.images.startIndex
) ?? 0
if index == thresholdIndex {
    viewModel.search()
}
```

- **RemoteImageView** — отдельный View, отвечающий за отображение и подгрузку картинок:

```
struct RemoteImageView: View {
    @StateObject private var loader = ImageLoader()
    let urlString: String
    ...
}
```

Внутри `RemoteImageView` — логика загрузки с кэшированием и показом плейсхолдера, если изображение ещё не загружено.

Преимущества реализации:

- Работает плавная прокрутка
- Загружаются изображения по мере прокрутки
- Нет лишних перерисовок благодаря `LazyVGrid`

2.1. Модель данных: ImageModel

Модель `ImageModel` — это структура, описывающая одно изображение, полученное из Unsplash API.

```
struct ImageModel: Codable {
    let id: String
    let urls: ImageURLs
    let description: String?

    struct ImageURLs: Codable {
        let small: String
        let regular: String
    }
}
```

Поскольку структура `ImageModel` соответствует формату JSON-ответа, она помечена протоколом `Codable`, что упрощает её декодирование/кодирование с помощью `JSONDecoder` / `JSONEncoder`.

2.2. Пагинация и динамический поиск: ViewModel

Вся логика по загрузке изображений сосредоточена в `ImageGalleryViewModel`. Это ключевая часть проекта.

Основные свойства:

- `images` — текущий массив изображений
- `query` — строка поиска
- `currentPage` — текущая страница поиска
- `canLoadMore` — флаг, можно ли ещё грузить
- `cacheOffset` — текущая позиция в кэше

- `isLoadingFromNetwork / isLoadingFromCache` — блокировка одновременной загрузки. Если вдруг мы больше не можем получать картинки из сети, то начинаем доставать их из кэша для возможности бесконечной прокрутки.

Метод `search(reset: Bool)`

```
func search(reset: Bool = false) {
    if reset { ... } // сбрасываем страницу
    guard !isLoadingFromNetwork && !isLoadingFromCache else { return }
    ...

    let trimmedQuery = query.trimmingCharacters(in: .whitespacesAndNewline
s)
    // если не получили images (например нет сети), идем в cache
    if trimmedQuery.isEmpty && images.isEmpty {
        loadFromCache()
        return
    }

    if trimmedQuery.isEmpty && !canLoadMore {
        loadFromCache()
    } else {
        loadFromNetwork(query: trimmedQuery)
    }
}
```

Метод `loadFromNetwork(query: String)`

Обращается к `NetworkService` и вызывает нужный метод в зависимости от наличия строки поиска.

```
if query.isEmpty {
        NetworkService.shared.fetchRandomImages(count: Constants.cachePag
eSize) { [weak self] result in
            self?.handleNetworkResult(result)
        }
```

```
} else {
        NetworkService.shared.searchImages(query: query, page: currentPage)
{ [weak self] result in
            self?.handleNetworkResult(result)
    }
}
```

Метод `handleNetworkResult(_ result: Result<[ImageModel], Error>)` обрабатывает результат:

- Фильтрует дубликаты по `id`
- Добавляет новые изображения в массив
- Кэширует обновлённый массив через `ImageCacher.save`
- При ошибках — переходит к кэшу

Метод `loadFromCache()`

```
ImageCacher.load(offset: cacheOffset, limit: Constants.cachePageSize) {
[weak self] cachedImages in
        guard let self = self else { return }
        let imagesToAppend = cachedImages
    if imagesToAppend.isEmpty {
            self.cacheOffset = 0
        ImageCacher.load(offset: self.cacheOffset, limit: Constants.cachePageSi
ze) { fallbackImages in
                self.appendCacheImages(fallbackImages)
            self.isLoadingFromCache = false
        }
    } else {
            self.appendCacheImages(imagesToAppend)
        self.isLoadingFromCache = false
    }
}
```

Если изображения из кэша закончились — начинается показ заново с начала.

3. Кэширование: ImageCacher

`ImageCacher` отвечает за сохранение и чтение изображений с использованием `UserDefaults`. Для оптимизации производительности были выделены статические свойства `JSONEncoder` и `JSONDecoder`, так как они могут использоваться повторно без необходимости пересоздания.

```
final class ImageCacher {
    static let encoder = JSONEncoder()
    static let decoder = JSONDecoder()
```

Сохранение:

```
static func save(_ images: [ImageModel]) {
    DispatchQueue.global(qos: .utility).async {
        do {
            let data = try encoder.encode(images)
            UserDefaults.standard.set(data, forKey: Constants.key)
        } catch {
            print("\(Constants.cacheError) \(error)")
        }
    }
}
```

- Кодировка и сохранение выполняются в фоновом потоке с приоритетом `utility`.

Загрузка:

```
static func load(offset: Int = 0, limit: Int = Constants.loadLimit,
                 completion: @escaping ([ImageModel]) -> Void) {
    DispatchQueue.global(qos: .userInitiated).async {
        guard let data = UserDefaults.standard.data(forKey: Constants.key) else {
            DispatchQueue.main.async { completion([]) }
```

```
            return
        }
        do {
            let allImages = try decoder.decode([ImageModel].self, from: data)
            let slice = Array(allImages.dropFirst(offset).prefix(limit))
            DispatchQueue.main.async {
                completion(slice)
            }
        } catch {
            print("\(Constants.readError) \(error)")
            DispatchQueue.main.async {
                completion([])
            }
        }
    }
}
```

- Декодирование и выборка нужного диапазона также происходят в фоне (с приоритетом `userInitiated`), но результат передаётся в основной поток.

4. Асинхронная загрузка изображений: ImageLoader

`ImageLoader` отвечает за подгрузку изображений из сети и их кэширование в оперативной памяти через `NSCache`.

```
final class ImageLoader: ObservableObject {
    @Published var image: UIImage?

    private static let imageCache = NSCache<NSString, UIImage>()
    private var urlString: String?

    func loadImage(from urlString: String) {
        self.urlString = urlString
```

```
        if let cachedImage = ImageLoader.imageCache.object(forKey: urlString a
s NSString) {
            self.image = cachedImage
            return
        }

        guard let url = URL(string: urlString) else { return }

        URLSession.shared.dataTask(with: url) { data, _, error in
            guard
                let data = data,
                let uiImage = UIImage(data: data),
                error == nil
            else { return }

            DispatchQueue.main.async {
                ImageLoader.imageCache.setObject(uiImage, forKey: urlString as N
SString)
                if self.urlString == urlString {
                    self.image = uiImage
                }
            }
        }.resume()
    }
}
```

- Проверяется наличие изображения в `NSCache`, и если оно найдено — оно сразу отображается
- В противном случае происходит асинхронная загрузка по сети
- При загрузке важно проверять, что `urlString` не изменилась, чтобы избежать гонок данных при повторном использовании `View`

5. Работа с API: NetworkService

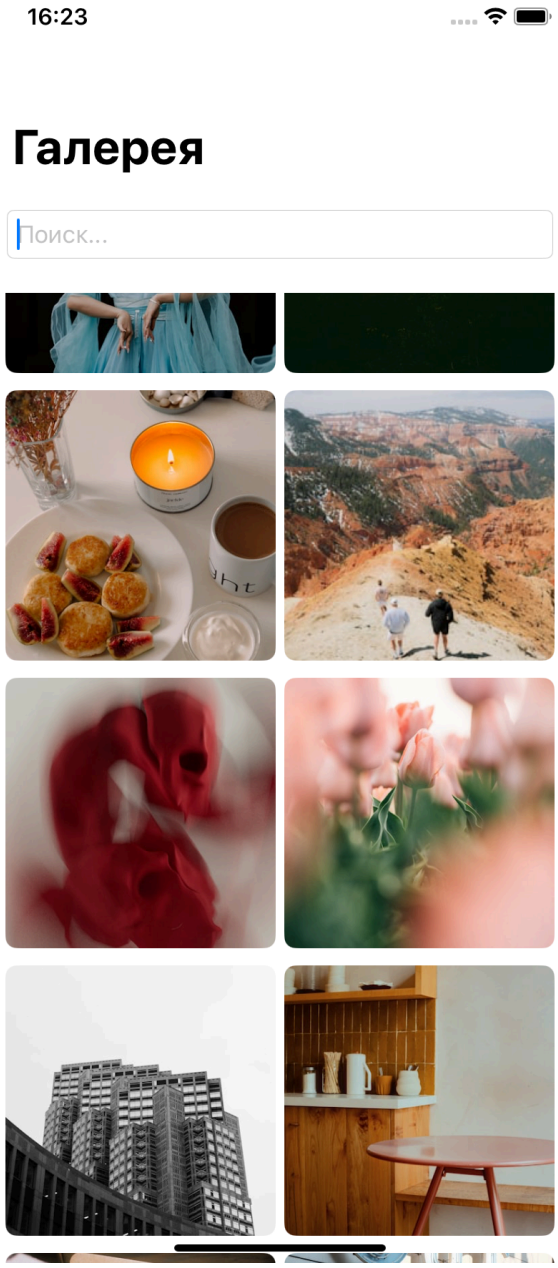
В данной реализации бесконечного скролла используется загрузка из сети, однако вы в своем проекте можете использовать другие источники данных, поэтому не будем заострять особое внимание на этом пункте. `NetworkService` выполняет два метода: получение случайных картинок или поиск по запросу `query`.

В `NetworkService` есть два метода:

- `fetchRandomImages()` — случайные картинки (для пустого поиска)
- `searchImages(query:page:)` — поиск по ключевому слову

Результат

Вы реализовали бесконечный скролл с возможностью поиска, пагинацией и локальным кэшированием. Даже при отсутствии интернета приложение продолжит работать с уже загруженными данными.


16:23
Галерея
Поиск...


16:22
Галерея
cats