원본글 주소: https://www.raywenderlich.com/5293-operation-and-operationqueue-tutorial-in-swift

Operation and OperationQueue Tutorial in Swift

이 자습서에서는 동시(Concurrent) Operation을 사용하여, Operation and OperationQueue를 다루고, 사용자에게 반응형 인터페이스를 제공하는 앱을 만듭니다.

iOS 또는 Mac 앱에서 버튼을 누르거나 텍스트를 입력하였을 때, 사용자 인터페이스가 응답하지 않는 것은 모두에게 갑작스런 경험입니다.

Mac에서는 사용자가 UI와 다시 상호 작용할 수 있을 때까지, 인디케이터 회전을 잠시 쳐다 봐야합니다. iOS 앱에서, 사용자는 앱이 터치에 즉시 응답하기를 기대합니다. 응답이 없는 앱은 불편하고 느리게 느껴지며, 일반적으로 리뷰가 좋지 않습니다.

앱 응답성을 유지하는 것은 말은 쉽습니다. 일단, 앱이 몇개 이상의 작업을 수행해야하는 경우 상황이 빠르게 복잡해집니다. 메인 런 루프에서 많은 작업을 수행하면서, 반응성이 좋은 UI를 제공하는 것은 불가능합니다.

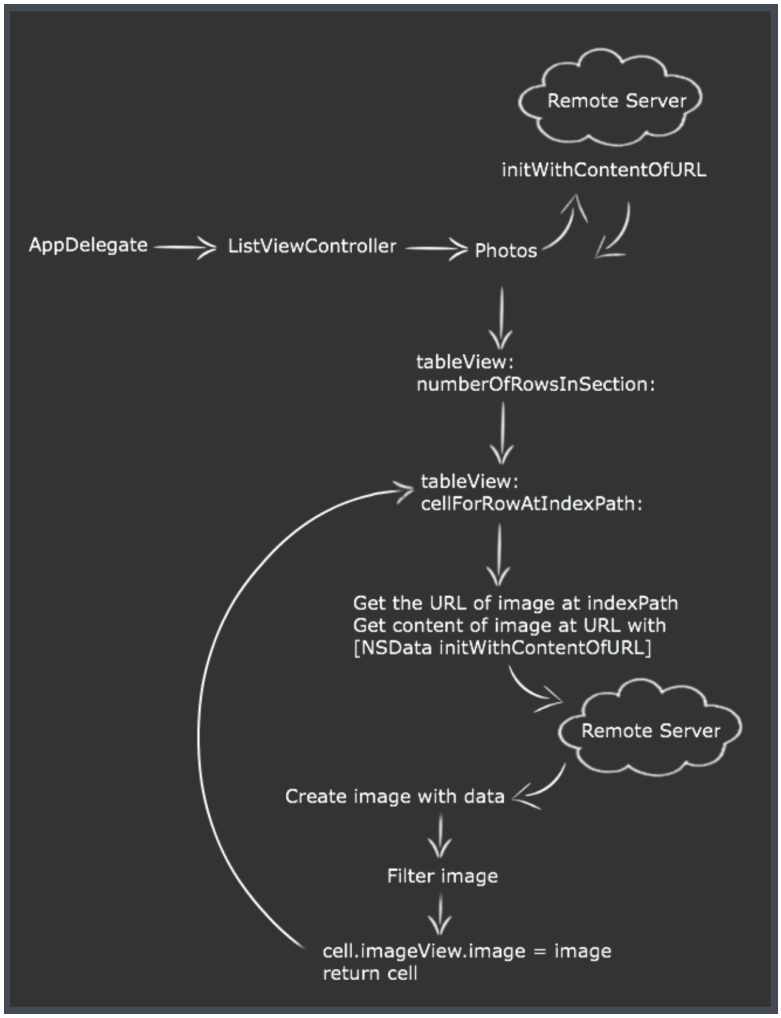
불쌍한 개발자는 무엇을 해야합니까? 해결책은 동시성을 통해 메인스레드에서 작업을 이동시키는 것입니다. 동시성은 애플리케이션이 동시에 여러 스트림(또는 스레드)의 작업을 실행함을 의미합니다. 이렇게 하면 작업을 수행하는 동안 사용자 인터페이스가 계속 반응합니다.

iOS에서 동시에 작업을 수행하는 한 가지 방법은 "Operation"및 "OperationQueue"클래스를 사용하는 것입니다. 이 튜토리얼에서는, 사용법을 배웁니다! 동시성을 전혀 사용하지 않는 앱으로 시작하므로 매우 느리고 응답하지 않는 것처럼 보입니다. 그런 다음 앱을 재작업하여 동시(Concurrent) 오퍼레이션을 추가하고 사용자에게보다 반응적인 인터페이스를 제공합니다.

Getting Started(시작하기)

이 학습서의 샘플 프로젝트의 전체 목표는 필터링된 이미지의 테이블뷰를 표시하는 것입니다. 인터넷에서 이미지를 다운로드하고, 필터를 적용한 다음, 테이블뷰에 표시합니다.

앱 모델의 개략도는 다음과 같습니다.

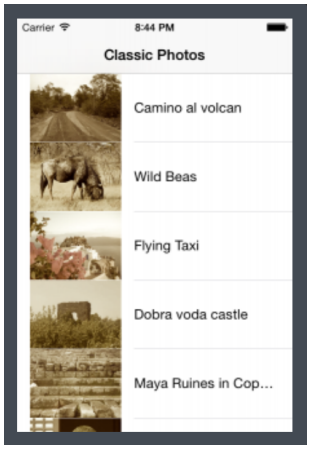


A First Try(첫 시도)

이 튜토리얼의 맨 위 또는 맨 아래에 있는 **자료 다운로드** 버튼을 사용하여 스타터 프로젝트를 다운로드하십시오. 이 자습서에서 작업 할 프로젝트의 첫 번째 버전입니다.

> 참고 : 모든 이미지는 stock.xchng에서 가져온 것입니다. 데이터 원본의 일부 이미지 이름이 의도적으로 잘못 지정되었으므로 이미지가 다운로드되지 않아 실패 사례가 발생하는 경우가 있습니다.

프로젝트를 빌드하고 실행하면 결국 사진 목록과 함께 앱이 실행됩니다. 목록을 스크롤 해보십시오. 고통스럽지 않습니까?



모든 동작은 **ListViewController.swift**에서 발생하며 대부분

tableView(_ : cellForRowAtIndexPath :) 안에 있습니다.

이 방법을 살펴보고 집중적인 두 가지 사항이 있습니다.

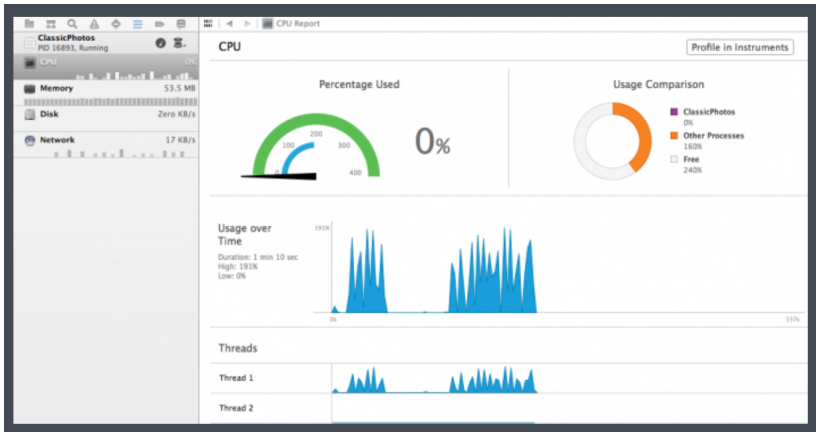
1. **웹에서 이미지 데이터를 로드합니다.** 이 작업은 쉬운 일임에도 불구하고, 앱은 계속 다운로드하기 전에 다운로드가 완료 될 때까지 기다려야합니다.

2. **코어 이미지를 사용하여 이미지를 필터링합니다.** 이 메서드는 이미지에 세피아 필터를 적용합니다. 코어 이미지 필터에 대한 자세한 내용은 Swift에서 코어 이미지 시작을 확인하십시오. (https://www.raywenderlich.com/2305-core-image-tutorial-getting-started)

또한 웹이 처음 요청 될 때 웹에서 사진 목록을 로드합니다.

```
lazy var photos = NSDictionary(contentsOf:dataSourceURL)!
```

이 모든 작업은 애플리케이션의 메인스레드에서 수행됩니다. 메인 스레드는 유저 인터렉션을 담당하므로 웹에서 데이터를 다운로드하고 이미지를 필터링하는데 바쁘게 하면, 앱의 응답성이 저하됩니다. Xcode의 게이지뷰를 사용하여 이에 대한 간단한 개요를 얻을 수 있습니다. **디버그 탐색기(Command-7)**를 표시 한 다음 앱이 실행되는 동안 CPU를 선택하여 게이지뷰로 이동할 수 있습니다.



앱의 메인스레드 인 스레드1에서 모든 스파이크를 볼 수 있습니다. 더 자세한 정보는 인스트러먼트에서 앱을 실행할 수 있지만 다른 튜토리얼입니다. :]
(https://www.raywenderlich.com/397-instruments-tutorial-with-swift-getting-started)

이제 사용자 경험을 향상시킬 수 있는 방법에 대해 생각할 때입니다.

Task, Threads and Processes(작업, 스레드 및 프로세스)

계속 진행하기 전에 이해해야 할, 몇 가지 기술적 개념이 있습니다. 주요 용어는 다음과 같습니다.

▶**작업(Task)** : 수행해야하는 간단한 단일 작업입니다.

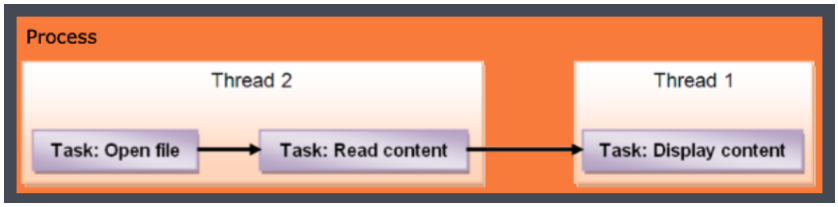
▶**스레드(Thread)** : 운영 체제에서 제공하는 메커니즘으로 단일 애플리케이션 내에서 여러 명령 세트를 동시에 작동 할 수 있습니다.

▶**프로세스(Process)** : 실행 가능한 코드 덩어리로 여러 스레드로 구성 될 수 있습니다.

참고 : iOS 및 macOS에서 스레딩 기능은 POSIX Threads API (또는 pthreads)에서 제공하며 운영 체제의 일부입니다. 이것은 매우 낮은 수준의 것들이므로 실수를 저지르기 쉽다는 것을 알게 될 것입니다. 아마도 스레드에서 가장 나쁜 점은 그 실수를 찾기가 매우 어려울 수 있다는 것입니다!

Foundation 프레임 워크에는 처리하기가 훨씬 쉬운 **Thread**라는 클래스가 포함되어 있지만, Thread를 사용하여 여러 스레드를 관리하는 것은 여전히 어려운 일입니다. Operation 및 OperationQueue는 여러 스레드를 처리하는 프로세스를 크게 단순화 한 고급 클래스입니다.

이 다이어그램에서 프로세스(process), 스레드(threads) 및 작업(tasks) 간의 관계를 볼 수 있습니다.



보시다시피, 프로세스에는 여러 실행 스레드가 포함될 수 있으며, 각 스레드는 한 번에 하나씩 여러 작업을 수행 할 수 있습니다.

이 다이어그램에서 스레드 2는 파일 읽기 작업을 수행하고 스레드 1은 사용자 인터페이스 관련 코드를 수행합니다. 이것은 iOS에서 코드를 구성하는 방법과 매우 유사합니다. 메인 스레드는 사용자 인터페이스와 관련된 작업을 수행하고, 두번째 스레드는 파일 읽기, 네트워크 액세스 등과 같은 느리거나 오래 실행되는 작업을 수행합니다.

오퍼레이션 VS 그랜드 센트럴 디스패치 (GCD)

GCD(Grand Central Dispatch)에 대해 들어 보셨을 것입니다. 간단히 말해서, GCD는 언어 기능, 런타임 라이브러리 및 시스템 향상으로 구성되어 iOS 및 macOS에서 멀티 코어 하드웨어의 동시성(Concurrency)을 지원하기 위해 체계적이고 포괄적인 개선 기능을 제공합니다. GCD에 대한 자세한 내용은 Grand Central Dispatch Tutorial을 참조하십시오.
(https://www.raywenderlich.com/5370-grand-central-dispatch-tutorial-for-swift-4-part-1-2)

Operation 및 OperationQueue는 GCD를 기반으로 구축됩니다. 매우 일반적인 규칙으로, Apple은 최고 수준의 추상화를 사용하고, 필요하다고 표시되면 더 낮은 수준으로 드롭 다운하도록 권장합니다.

다음은 GCD 또는 오퍼레이션을 언제 어디서 사용할지 결정하는 데 도움이 되는 두 가지를 간단히 비교 한 것입니다.

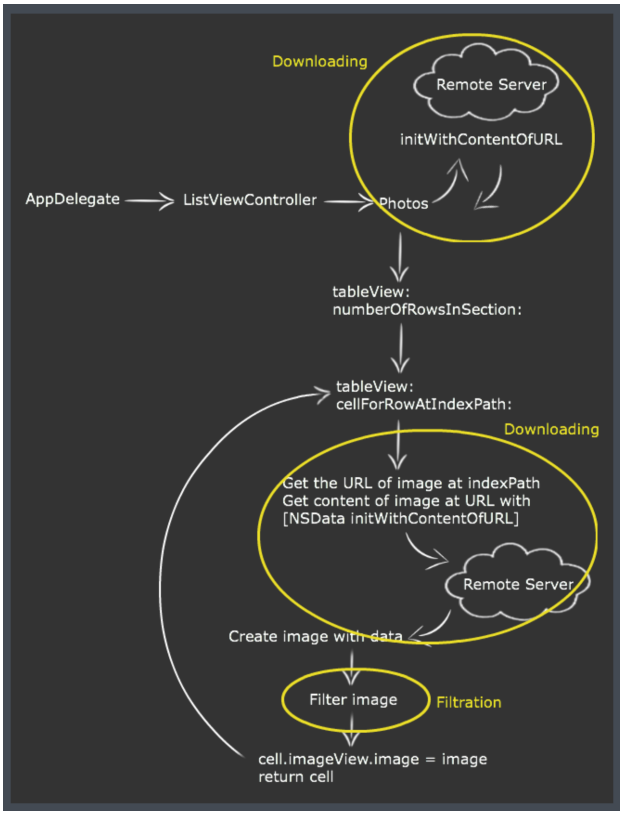
▶**GCD**는 동시에 실행될 작업 단위를 나타내는 간단한 방법입니다. 이러한 작업 단위를 스케줄링 하지는 않습니다. 시스템이 당신을 위해 스케줄링하고 관리합니다. 블록 사이에 의존성을 추가하는 것은 골치 아픈 일이 될 수 있습니다. 블록을 취소(canceling)하거나, 일시 중단(suspending)하면 개발자로서 추가 작업이 발생합니다!

▶**오퍼레이션**은 GCD에 비해, 약간의 머리쓰는 일을 추가하지만, 다양한 오퍼레이션간에 종속성을 추가하고 재사용, 취소 또는 일시중지(suspend) 할 수 있습니다.

이 튜토리얼에서는 테이블뷰를 다루기 때문에 "오퍼레이션"을 사용하며, 성능 및 전력 소비를 위해 사용자가 해당 이미지를 화면 밖으로 스크롤 한 경우 특정 이미지에 대한 오퍼레이션을 취소 할 수 있어야합니다. 오퍼레이션이 백그라운드 스레드에 있더라도 큐에 대기중인 수십 개가 있으면 성능이 여전히 저하됩니다.

Refined App Model(개선된 앱 모델)

앞선 논-스레드 모델을 개선 할 때입니다! 앞선 모델을 자세히 살펴보면 개선할 수있는 스레드가-지저분한(bogging) 영역이 세 개 있음을 알 수 있습니다. 이 세 영역을 분리하여 별도의 스레드에 배치하면 메인 스레드가 완화되고, 유저 인터렉션에 응답성있게 반응할 수 있습니다.



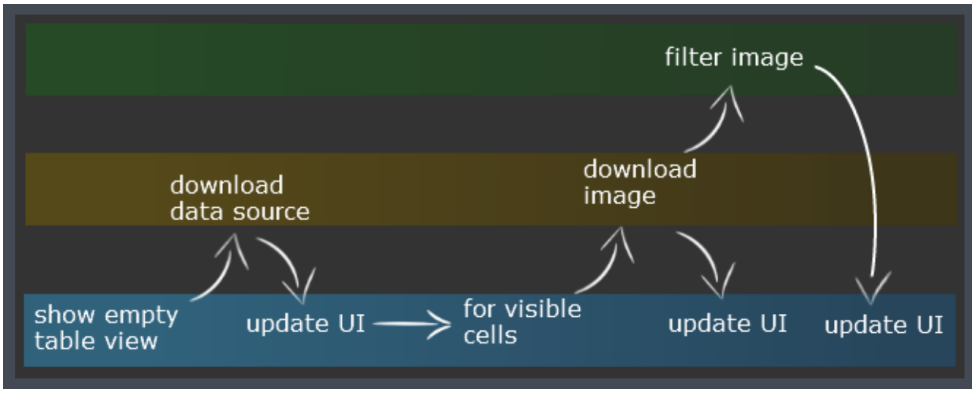
애플리케이션의 병목 현상을 제거하려면, 사용자 상호 작용에 응답하기위한 스레드, 데이터 소스 및 이미지 다운로드 전용 스레드 및 이미지 필터링을 수행하기 위한 스레드가 필요합니다. 새 모델에서, 앱은 메인 스레드에서 시작하여 빈 테이블뷰를 로드합니다. 동시에 앱은 두 번째 스레드를 시작하여 데이터 소스를 다운로드합니다.

일단 데이터 소스가 다운로드되면, 테이블뷰에 다시 로드하도록 지시합니다. 이것은 사용자 인터페이스와 관련되어 있기 때문에 메인 스레드에서 수행되어야합니다. 이 시점에서 테이블뷰는 행 수를 알고 표시해야하는 이미지의 URL을 알고 있지만, 실제 이미지는 아직 없습니다! 이 시점에서 모든 이미지를 즉시 다운로드하기 시작하면 한 번에 모든 이미지가 필요하지 않으므로 매우 비효율적입니다!

이것을 개선하기 위해 무엇을 할 수 있습니까?

더 나은 모델은 각 행이 화면에 보이는 이미지 다운로드를 시작하는 것입니다. 따라서 코드는 먼저 테이블뷰에 어떤 행이 표시되는지 물어보고, 다운로드 작업을 시작합니다. 마찬가지로 이미지가 완전히 다운로드 될 때까지 이미지 필터링 작업을 시작할 수 없습니다. 따라서 필터링되지 않은 이미지의 기다림이 처리 될 때까지, 앱은 이미지 필터링 작업을 시작하지 않아야합니다.

앱의 응답성을 높이기 위해 코드는 한번 이미지가 다운로드되면, 즉시 이미지를 표시합니다. 그런 다음 이미지 필터링을 시작한 다음, UI를 업데이트하여 필터링된 이미지를 표시합니다. 아래 다이어그램은 이에 대한 개략적인 제어 흐름을 보여줍니다.



이러한 목표를 달성하려면 이미지 다운로드 중인지, 다운로드 됐는지 여부 또는 필터링 여부를 추적해야합니다. 또한 각 오퍼레이션의 상태와 유형을 추적해야 사용자가 스크롤 할 때 각각의 작업을 취소, 일시 중지 또는 재개 할 수 있습니다.

괜찮아! 이제 코딩 할 준비가 되었습니다!

Xcode에서, **PhotoOperations.swift**라는 새 **Swift 파일**을 프로젝트에 추가하십시오. 다음 코드를 추가하십시오 :

```
import UIKit

// This enum contains all the possible states a photo record can be in
enum PhotoRecordState{
  case new, downloaded, filtered, failed
}

class PhotoRecord{
  let name: String
  let url: URL
  var state = PhotoRecordState.new
  var image = UIImage(named: "Placeholder")

  init(name:String, url:URL) {
    self.name = name
    self.url = url
  }
}
```

이 간단한 클래스는 앱에 표시되는 각 사진과 현재 상태(디폴트는 .new로 설정된)를 나타냅니다. 이미지는 디폴트로 플레이스홀더로 설정됩니다.

각 오퍼레이션의 상태를 추적하려면, 별도의 클래스가 필요합니다. **PhotoOperations.swift**의 끝에 다음 정의를 추가하십시오.

```
class PendingOperations{
  lazy var downloadsInProgress: [IndexPath: Operation] = [:]
  lazy var downloadQueue: OperationQueue = {
    var queue = OperationQueue()
    queue.name = "Download queue"
    queue.maxConcurrentOperationCount = 1
    return queue
  }()

  lazy var filtrationsInProgress: [IndexPath: Operation] = [:]
  lazy var filtrationQueue: OperationQueue = {
    var queue = OperationQueue()
    queue.name = "Image Filtration queue"
    queue.maxConcurrentOperationCount = 1
    return queue
  }()
}
```

이 클래스에는 테이블의 각 행에 대한 활성(active) 및 보류중인(pending) 다운로드 및 필터 오퍼레이션을 추적하기위한 두 개의 딕셔너리와 각 오퍼레이션 유형에 대한 오퍼레이션큐가 있습니다.

모든 값은 지연되어(lazily) 생성되며 처음 액세스 할 때까지 초기화되지 않습니다. 앱의 성능이 향상됩니다.

보다시피 OperationQueue 생성은 매우 간단합니다. 큐 이름을 지정하면 계측기(instruments) 또는 디버거에 이름이 표시되므로 디버깅에 도움이됩니다. 이 튜토리얼에서는 maxConcurrentOperationCount를 1로 설정하여 작업을 하나씩 완료하는 것을 볼 수 있습니다. 이 부분을 제외하고, 큐에서 한 번에 처리 할 수 있는 작업 수를 결정하도록 허용하면 성능이 더욱 향상됩니다.

큐는 한 번에 실행할 수있는 작업 수를 어떻게 결정합니까? 그건 좋은 질문입니다! **하드웨어에 따라 다릅니다.** 디폴트로 OperationQueue는 배후에서 약간의 계산을 수행하고 실행중인 특정 플랫폼에 가장 적합한 것을 결정하고, 가능한 최대 스레드 수를 시작합니다.

다음 예를 고려하십시오. 시스템이 유휴 상태이고 사용 가능한 많은 자원이 있다고 가정하십시오. 이 경우 큐는 8개의 실제 동시적으로(simultaneous) 실행하는 스레드를 시작할 수 있습니다. 다음에 프로그램을 실행하면, 새로 시

작한 오퍼레이션과 관련없이, 시스템이 현재 리소스를 소비하고 있는 오퍼레이션으로 인해 바쁠 수 있습니다. 이번에는 큐가 두 개의 동시(simultaneous) 스레드만 시작할 수 있습니다. 이 앱에서 최대 동시 작업 수(maximum concurrent operations)를 설정 했으므로, 오직 한 번에 하나의 오퍼레이션만 발생합니다.

> 참고 : 왜 모든 활성(active) 및 보류중인(pending) 작업을 추적해야하는지 궁금할 것입니다. 큐에는 오퍼레이션 배열을 반환하는 operations메서드가 있으므로, 왜 사용하지 않습니까? 이 프로젝트에서는 그렇게 효율적이지 않을 것입니다. 어떤 오퍼레이션이 어떤 테이블뷰 행과 관련되어 있는지 추적해야하며, 필요할 때마다 배열을 반복해야합니다. indexpath를 키로하여 딕셔너리에 저장하면 조회가 빠르고 효율적입니다.

이제 다운로드 및 필터링 오퍼레이션을 처리해야합니다. **PhotoOperations.swift**의 끝에 다음 코드를 추가하십시오:

```
class ImageDownloader: Operation{
  //1
  let photoRecord: PhotoRecord

  //2
  init(_ photoRecord: PhotoRecord) {
    self.photoRecord = photoRecord
  }

  //3
  override func main() {
    //4
    if isCancelled {
      return
    }

    //5
    guard let imageData = try? Data(contentsOf: photoRecord.url) else { return }

    //6
    if isCancelled {
      return
    }

    //7
    if !imageData.isEmpty {
      photoRecord.image = UIImage(data:imageData)
      photoRecord.state = .downloaded
    } else {
      photoRecord.state = .failed
      photoRecord.image = UIImage(named: "Failed")
    }
  }
}
```

오퍼레이션은 서브클래싱을 위해 설계된 추상 클래스입니다. 각 서브클래스는 앞의 다이어그램에 표시된 특정 **작업(task)**을 나타냅니다.

위 코드에서 번호가 매겨진 각 주석에서 발생하는 상황은 다음과 같습니다.

1. 오퍼레이션과 관련된 PhotoRecord 객체에 대한 상수 참조를 추가하십시오.

2. 사진 기록을 전달할 수 있는 지정된(designated) 이니셜라이저를 만듭니다.

3. main()은 실제로 작업을 수행하기 위해 Operation 서브클래스에서 override하는 메소드입니다.

4. 시작하기 전에 취소를 확인하십시오. 장시간 또는 집중적인 오퍼레이션을 시도하기 전에 작업이 취소되었는지 정기적으로 확인해야합니다.

5. 이미지 데이터를 다운로드하십시오.

6. 취소를 다시 확인하십시오.

7. 데이터가 있으면, 이미지 객체를 만들어 레코드에 추가하고, 상태(state)도 함께 이동하십시오. 데이터가 없으면, 레코드를 실패로 표시하고 적절한 이미지를 설정하십시오.

다음으로 이미지 필터링을 처리하기위한 다른 오퍼레이션을 만듭니다.
PhotoOperations.swift의 끝에 다음 코드를 추가하십시오 :

```
class ImageFiltration: Operation{
  let photoRecord: PhotoRecord

  init(_ photoRecord: PhotoRecord) {
    self.photoRecord = photoRecord
  }

  override func main () {
    if isCancelled {
        return
    }

    guard self.photoRecord.state == .downloaded else {
      return
    }

    if let image = photoRecord.image,
       let filteredImage = applySepiaFilter(image) {
      photoRecord.image = filteredImage
      photoRecord.state = .filtered
    }
  }
}
```

이미지를 다운로드하는 대신 필터를 이미지에 적용하는 것 (아직 구현되지 않은 방법을 사용하므로 컴파일러 오류)을 제외하고는 다운로드 오퍼레이션과 매우 유사합니다.

누락된 이미지 필터 메소드를 ImageFiltration 클래스에 추가하십시오.

```
func applySepiaFilter(_ image: UIImage) -> UIImage? {
  guard let data = UIImagePNGRepresentation(image) else { return nil }
  let inputImage = CIImage(data: data)

  if isCancelled {
    return nil
  }

  let context = CIContext(options: nil)

  guard let filter = CIFilter(name: "CISepiaTone") else { return nil }
  filter.setValue(inputImage, forKey: kCIInputImageKey)
  filter.setValue(0.8, forKey: "inputIntensity")

  if isCancelled {
    return nil
  }

  guard
    let outputImage = filter.outputImage,
    let outImage = context.createCGImage(outputImage, from: outputImage.extent)
  else {
    return nil
  }

  return UIImage(cgImage: outImage)
}
```

이미지 필터링은 이전에 ListViewController에서 사용 된 것과 동일한 구현입니다. 백그라운드에서 별도의 오퍼레이션으로 수행 할 수 있도록 여기로 이동되었습니다. 다시 한번 취소(cancellation)를 매우 자주 확인해야합니다. 값비싼 메소드 호출 전후에, 이를 수행하는 것이 좋습니다. 일단 필터링이 완료되면, photo record 인스턴스의 값을 설정합니다.

Great! 이제 오퍼레이션을 백그라운드 작업으로 처리하는 데 필요한 모든 도구(tools)와 기초(foundation)가 있습니다. 이제 뷰컨트롤러로 돌아가서, 이러한 새로운 이점을 모두 활용할 수 있도록 수정해야합니다.

ListViewController.swift로 전환하고 lazy var photos 속성 선언을 삭제하십시오. 대신 다음 선언을 추가하십시오.

```
var photos: [PhotoRecord] = []
let pendingOperations = PendingOperations()
```

이러한 속성은 작업을 관리하기 위해 PhotoRecord 객체 배열과 PendingOperations 객체를 보유합니다.

사진 속성 목록을 다운로드하려면 클래스에 새 메소드를 추가하십시오.:

```
func fetchPhotoDetails() {
  let request = URLRequest(url: dataSourceURL)
  UIApplication.shared.isNetworkActivityIndicatorVisible = true

  // 1
  let task = URLSession(configuration: .default).dataTask(with: request) { data, response, error in

    // 2
    let alertController = UIAlertController(title: "Oops!",
                                            message: "There was an error fetching photo details.",
                                            preferredStyle: .alert)
    let okAction = UIAlertAction(title: "OK", style: .default)
    alertController.addAction(okAction)

    if let data = data {
      do {
        // 3
        let datasourceDictionary =
          try PropertyListSerialization.propertyList(from: data,
                                                     options: [],
                                                     format: nil) as! [String: String]

        // 4
        for (name, value) in datasourceDictionary {
          let url = URL(string: value)
          if let url = url {
            let photoRecord = PhotoRecord(name: name, url: url)
            self.photos.append(photoRecord)
          }
        }

        // 5
        DispatchQueue.main.async {
          UIApplication.shared.isNetworkActivityIndicatorVisible = false
          self.tableView.reloadData()
        }
        // 6
      } catch {
        DispatchQueue.main.async {
          self.present(alertController, animated: true, completion: nil)
        }
      }
    }

    // 6
    if error != nil {
      DispatchQueue.main.async {
        UIApplication.shared.isNetworkActivityIndicatorVisible = false
        self.present(alertController, animated: true, completion: nil)
      }
    }
  }
  // 7
  task.resume()
}
```

이것이 하는 일은 다음과 같습니다.

1. URLSession data task를 작성하여 백그라운드 스레드에서 이미지의 특성 목록(property list of images)을 다운로드하십시오.

2. 오류 발생시 사용할 UIAlertController를 구성하십시오.

3. 요청이 성공하면, 속성 목록(property list)에서 딕셔너리를 작성하십시오. 딕셔너리는 이미지 이름을 키(key)로 사용하고 URL을 값으로 사용합니다.

4. 딕셔너리에서 PhotoRecord 객체의 배열을 만듭니다.

5. 메인 스레드로 돌아가 테이블뷰를 다시로드하고 이미지를 표시합니다.

6. 오류 발생시, 얼럿 컨트롤러를 표시합니다. URLSession 작업은 백그라운드 스레드에서 실행되며, 화면의 메시지 표시는 메인 스레드에서 수행해야합니다.

7. 다운로드 작업을 실행하십시오.

viewDidLoad()의 끝에서 새 메소드를 호출하십시오.

```
fetchPhotoDetails()
```

다음으로 tableView(_ : cellForRowAtIndexPath :)를 찾으십시오. 컴파일러가 이에 대해 불평하기 때문에 쉽게 찾을 수 있습니다. 다음 구현으로 대체하십시오.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(withIdentifier: "CellIdentifier", for: indexPath)

  //1
  if cell.accessoryView == nil {
    let indicator = UIActivityIndicatorView(activityIndicatorStyle: .gray)
    cell.accessoryView = indicator
  }
  let indicator = cell.accessoryView as! UIActivityIndicatorView

  //2
  let photoDetails = photos[indexPath.row]

  //3
  cell.textLabel?.text = photoDetails.name
  cell.imageView?.image = photoDetails.image

  //4
  switch (photoDetails.state) {
  case .filtered:
    indicator.stopAnimating()
  case .failed:
    indicator.stopAnimating()
    cell.textLabel?.text = "Failed to load"
  case .new, .downloaded:
    indicator.startAnimating()
    startOperations(for: photoDetails, at: indexPath)
  }

  return cell
}
```

이것이 하는 일은 다음과 같습니다.

1. 사용자에게 피드백을 제공하려면 UIActivityIndicatorView를 작성하고 셀의 accessory view로 설정하십시오.

2. 데이터 소스에는 PhotoRecord의 인스턴스가 포함됩니다. 현재 indexPath를 기반으로 올바른 것을 가져옵니다.

3. 셀의 텍스트 레이블은 (거의) 항상 동일하며, 이미지는 처리 될 때 PhotoRecord에서 적절하게 설정되므로, 레코드 상태(state of the record)에 관계없이 여기에서 둘 다 설정할 수 있습니다.

4. record를 점검하십시오. activity indicator 및 텍스트를 적절하게 설정하고, 오퍼레이션을 시작하십시오(아직 구현되지 않음).

다음 메소드를 클래스에 추가하여 오퍼레이션을 시작하십시오.

```
func startOperations(for photoRecord: PhotoRecord, at indexPath: IndexPath) {
  switch (photoRecord.state) {
  case .new:
    startDownload(for: photoRecord, at: indexPath)
  case .downloaded:
    startFiltration(for: photoRecord, at: indexPath)
  default:
    NSLog("do nothing")
  }
}
```

여기에서는, 인덱스 경로와 함께 PhotoRecord 인스턴스를 전달합니다. 사진 기록(photo record)의 상태에 따라 다운로드 또는 필터 오퍼레이션을 시작합니다.

참고 : 이미지 다운로드 및 필터링 방법은, 이미지를 다운로드하는 동안 사용자가 스크롤 할 수 있으며 이미지 필터를 아직 적용하지 않았을 수 있으므로, 별도로 구현됩니다. 다음에 사용자가 같은 행을 방문하면, 이미지를 다시 다운로드 할 필요가 없습니다. 이미지 필터 만 적용하면됩니다. 효율성이 짱이야! :]

이제 위의 메소드에서 호출한 메소드를 구현해야합니다. 오퍼레이션을 추적하기 위해, 커스텀 클래스 PendingOperations를 작성했음을 기억하십시오. 이제 실제로 사용하십시오! 클래스에 다음 메소드를 추가하십시오.

```
func startDownload(for photoRecord: PhotoRecord, at indexPath: IndexPath) {
  //1
  guard pendingOperations.downloadsInProgress[indexPath] == nil else {
    return
  }

  //2
  let downloader = ImageDownloader(photoRecord)

  //3
  downloader.completionBlock = {
    if downloader.isCancelled {
      return
    }

    DispatchQueue.main.async {
      self.pendingOperations.downloadsInProgress.removeValue(forKey: indexPath)
      self.tableView.reloadRows(at: [indexPath], with: .fade)
    }
  }

  //4
  pendingOperations.downloadsInProgress[indexPath] = downloader

  //5
  pendingOperations.downloadQueue.addOperation(downloader)
}

func startFiltration(for photoRecord: PhotoRecord, at indexPath: IndexPath) {
  guard pendingOperations.filtrationsInProgress[indexPath] == nil else {
      return
  }

  let filterer = ImageFiltration(photoRecord)
  filterer.completionBlock = {
    if filterer.isCancelled {
      return
    }

    DispatchQueue.main.async {
      self.pendingOperations.filtrationsInProgress.removeValue(forKey: indexPath)
      self.tableView.reloadRows(at: [indexPath], with: .fade)
    }
  }

  pendingOperations.filtrationsInProgress[indexPath] = filterer
  pendingOperations.filtrationQueue.addOperation(filterer)
}
```

위 코드에서 진행중인 작업을 이해하기위한 빠른 목록은 다음과 같습니다.

1. 먼저 특정 indexPath를 확인하여 downloadsInProgress에 이미 오퍼레이션이 있는지 확인하십시오. 그렇다면 이 요청을 무시하십시오.

2. 그렇지 않은 경우 지정된 이니셜라이저(designated initializer)를 사용하여 ImageDownloader 인스턴스를 만듭니다.

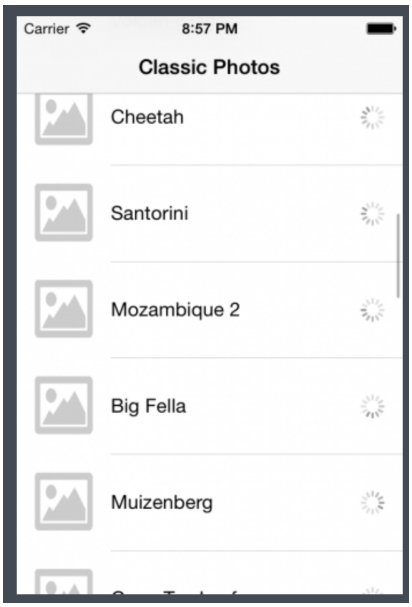
3. 오퍼레이션 완료시, 실행될 completion block을 추가하십시오. 이곳은 나머지 앱에 오퍼레이션이 완료되었음을 알리는 좋은 장소입니다. 오퍼레이션이 취소 된 경우에도, completion block이 실행되므로, 이 오퍼레이션을 수행하기 전에 이 속성을 확인해야합니다. 또한 completion block이 호출된 스레드를 보장 할 수 없으므로, GCD를 사용하여 메인 스레드에서 테이블뷰의 reload를 트리거해야합니다.

4. downloadsInProgress에 오퍼레이션을 추가하여 사물을 추적하는 데 도움이됩니다.

5. 다운로드 큐에 오퍼레이션을 추가하십시오. 이렇게 하면 실제로 이러한 오퍼레이션이 실행되기 시작합니다. 오퍼레이션을 추가하면 큐에서 일정을 관리합니다.

이미지를 필터링하는 메서드는 ImageFiltration 및 filtrationsInProgress를 사용하여 오퍼레이션을 추적한다는 점을 제외하고 동일한 패턴을 따릅니다. 연습으로, 이 코드 섹션에서 반복을 제거 할 수 있습니다. :]

이제 다했어요! 프로젝트가 완료되었습니다. 개선 사항을 확인하기 위해, 빌드하고 실행하십시오! 테이블뷰를 스크롤하면 앱이 더 이상 정지되지 않고, 이미지가 보여지게 될 때 이미지 다운로드 및 필터링을 시작합니다.



멋지지 않습니까? 애플리케이션의 반응 속도를 높이고, 사용자에게 훨씬 더 재미를 주기 위해 약간의 노력을 기울이는 방법을 알 수 있습니다!

Fine Tuning(미세 조정)

이 튜토리얼에서 먼 길을 왔습니다! 작은 프로젝트는 반응성이 좋으며, 원본 버전보다 많은 개선이 이루어졌습니다. 그러나 여전히 약간의 세부 사항이 남아 있습니다.

테이블뷰에서 스크롤 할 때, 해당 오프 스크린 셀이 여전히 다운로드 및 필터링되는 중임을 알 수 있습니다. 빠르게 스크롤하면 이미 아주 멀리 벗어난 앱에 표시되지 않는 셀에서 이미지를 다시 다운로드하여 필터링하는 작업이 바쁠 것입니다. 이상적으로는 앱에서 화면을 벗어난 셀의 필터링을 취소하고, 현재 표시된 셀에 우선순위를 지정해야합니다.

코드에 취소 조항을 넣지 않았습니까? 그렇습니다. 이제 사용 하셨을 것입니다. :]

ListViewController.swift를여십시오. tableView(_ : cellForRowAtIndexPath :) 구현으로 이동하여 다음과 같이 if 문에서 startOperationsForPhotoRecord에 대한 호출을 래핑하십시오.

```
if !tableView.isDragging && !tableView.isDecelerating {
  startOperations(for: photoDetails, at: indexPath)
}
```

테이블뷰가 스크롤되지 않는 경우에만, 테이블뷰에 오퍼레이션을 시작하도록 지시합니다. 이들은 실제로 UIScrollView의 속성이며 UITableView는 UIScrollView의 하위 클래스이기 때문에 테이블뷰는 이러한 속성을 자동으로 상속합니다.

다음으로, 다음 UIScrollView 델리게이트 메소드의 구현을 클래스에 추가하십시오.

```
override func scrollViewWillBeginDragging(_ scrollView: UIScrollView) {
  //1
  suspendAllOperations()
}

override func scrollViewDidEndDragging(_ scrollView: UIScrollView, willDecelerate decelerate: Bool) {
  // 2
  if !decelerate {
    loadImagesForOnscreenCells()
    resumeAllOperations()
  }
}

override func scrollViewDidEndDecelerating(_ scrollView: UIScrollView) {
  // 3
  loadImagesForOnscreenCells()
  resumeAllOperations()
}
```

위의 코드를 간략히 살펴보면 다음과 같습니다.

1. 사용자가 스크롤을 시작하자마자 모든 오퍼레이션을 일시 중단(suspend)하고 사용자가보고 싶은 내용을 살펴볼 수 있습니다. 곧 suspendAllOperations를 구현할 것입니다.

2. 감속 값(value of decelerate)이 false이면 사용자가 테이블뷰 끌기를 중지했음을 의미합니다. 따라서, 일시 중단된(suspended) 오퍼레이션을 재개하고, 화면을 벗어난(오프) 스크린 셀에 대한 오퍼레이션을 취소하고 온 스크린 셀에 대한 오퍼레이션을 시작하려고합니다. 잠시 동안 loadImagesForOnscreenCells 및 resumeAllOperations를 구현합니다.

3. 이 델리게이트 메서드는 테이블뷰가 스크롤을 중지했음을 알려주므로, #2에서와 동일하게 수행됩니다.

이제 누락 된 메소드의 구현을 **ListViewController.swift**에 추가하십시오.

```
func suspendAllOperations() {
  pendingOperations.downloadQueue.isSuspended = true
  pendingOperations.filtrationQueue.isSuspended = true
}

func resumeAllOperations() {
  pendingOperations.downloadQueue.isSuspended = false
  pendingOperations.filtrationQueue.isSuspended = false
}

func loadImagesForOnscreenCells() {
  //1
  if let pathsArray = tableView.indexPathsForVisibleRows {
    //2
    var allPendingOperations = Set(pendingOperations.downloadsInProgress.keys)
    allPendingOperations.formUnion(pendingOperations.filtrationsInProgress.keys)

    //3
    var toBeCancelled = allPendingOperations
    let visiblePaths = Set(pathsArray)
    toBeCancelled.subtract(visiblePaths)

    //4
    var toBeStarted = visiblePaths
    toBeStarted.subtract(allPendingOperations)

    // 5
    for indexPath in toBeCancelled {
      if let pendingDownload = pendingOperations.downloadsInProgress[indexPath] {
        pendingDownload.cancel()
      }
      pendingOperations.downloadsInProgress.removeValue(forKey: indexPath)
      if let pendingFiltration = pendingOperations.filtrationsInProgress[indexPath] {
        pendingFiltration.cancel()
      }
      pendingOperations.filtrationsInProgress.removeValue(forKey: indexPath)
    }

    // 6
    for indexPath in toBeStarted {
      let recordToProcess = photos[indexPath.row]
      startOperations(for: recordToProcess, at: indexPath)
    }
  }
}
```

suspendAllOperations() 및 resumeAllOperations()에는 간단한 구현이 있습니다. suspended 속성을 true로 설정하여 OperationQueue들은 일시 중지(suspended)될 수 있습니다. 큐의 모든 오퍼레이션이 일시 중지됩니다. 오퍼레이션을 개별적으로 일시 중지 할 수 없습니다.

loadImagesForOnscreenCells()는 조금 더 복잡합니다. 진행중인 작업은 다음과 같습니다.

1. 테이블뷰에서 현재 보이는 모든 행의 indexpath를 포함하는 배열로 시작하십시오.

2. 진행중인 모든 다운로드와 진행중인 모든 필터를 결합하여, 모든 보류중인(pending) 오퍼레이션 세트를 구성하십시오.

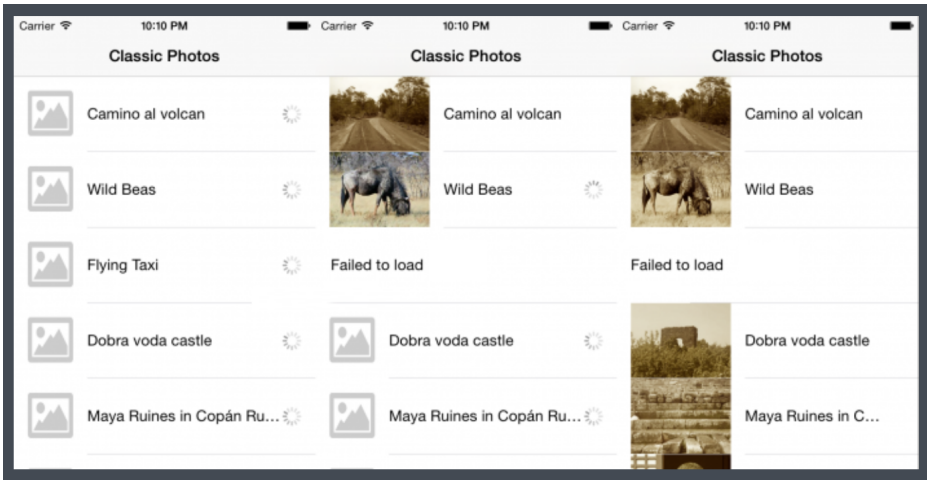
3. 취소 할 오퍼레이션의 모든 indexpath 세트를 구성하십시오. 모든 오퍼레이션으로 시작한 다음, 보이는 행의 indexpath를 제거하십시오. 스크린에서 벗어난 행과 관련된 일련의 오퍼레이션이 남습니다.

4. 오퍼레이션을 시작해야하는 일련의 indexpath를 구성하십시오. 표시된 모든 행을 indexpath로 시작한 다음 오퍼레이션이 이미 보류중(pending)인 행을 제거하십시오.

5. 취소 할 항목을 반복문으로 하고 취소 한 다음 PendingOperations에서 해당 참조를 제거하십시오.

6. 시작할 것들을 반복문으로 하고, 각각에 대해 startOperations(for : at :)를 호출하십시오.

빌드하고 실행하면 응답성이 뛰어나고 리소스 관리가 향상된 애플리케이션이 생겼습니다! 박수치세요!



테이블뷰 스크롤을 마치면 보이는 행의 이미지가 즉시 처리되기 시작합니다.

Where to Go From Here?(여기서 어디로 가야합니까?)

이 학습서의 맨 위 또는 맨 아래에 있는 **자료 다운로드** 단추를 사용하여 완료된 버전의 프로젝트를 다운로드 할 수 있습니다.

Operation 및 OperationQueue를 사용하여 소스 코드를 유지 관리하고 이해하기 쉽게 유지하면서 메인 스레드에서 떼내어 오래걸리는 계산을 이동하는 방법을 배웠습니다.

그러나 깊이 중첩된 블록과 같이 조심스럽게 멀티 스레딩을 사용하면 코드를 유지 관리해야하는 사람들이 프로젝트를 이해할 수 없게 될 수 있습니다. 스레드는 네트워크 속도가 느리거나 코드가 더 빠른 (또는 더 느린) 장치 또는 다른 코어 수를 가진 장치에서 실행될 때까지 나타나지 않는 미묘한 버그를 유발할 수 있습니다. 매우 신중하게 테스트하고 항상 인스트러먼트(또는 자체 관찰)를 사용하여 스레드를 도입하여 실제로 개선되었는지 확인하십시오.

여기서 다루지 않은 유용한 오퍼레이션 기능은 종속성입니다. 하나 이상의 다른 오퍼레이션에 따라 오퍼레이션을 수행 할 수 있습니다. 이 오퍼레이션은 종속된 오퍼레이션이 모두 완료 될 때까지 시작되지 않습니다. 예를 들면 다음과 같습니다.

```
// MyDownloadOperation is a subclass of Operation
let downloadOperation = MyDownloadOperation()
// MyFilterOperation  is a subclass of Operation
let filterOperation = MyFilterOperation()

filterOperation.addDependency(downloadOperation)
```

종속성을 제거하려면 다음을 수행하십시오.

```
filterOperation.removeDependency(downloadOperation)
```

종속성을 사용하여 이 프로젝트의 코드를 단순화하거나 개선 할 수 있습니까? 새로운 기술을 사용하고 사용해보십시오. :] 주의해야 할 중요한 사항은 종속된(앞단계의) 오퍼레이션이 취소되고 자연스럽게 완료되는 경우에도, 다음 오퍼레이션이 시작된다는 것입니다. 이를 명심해야합니다.

이 자습서 또는 일반적인 작업에 대한 의견이나 질문이 있으면 아래 포럼 토론에 참여하십시오!

원본글 주소: https://www.raywenderlich.com/5293-operation-and-operationqueue-tutorial-in-swift