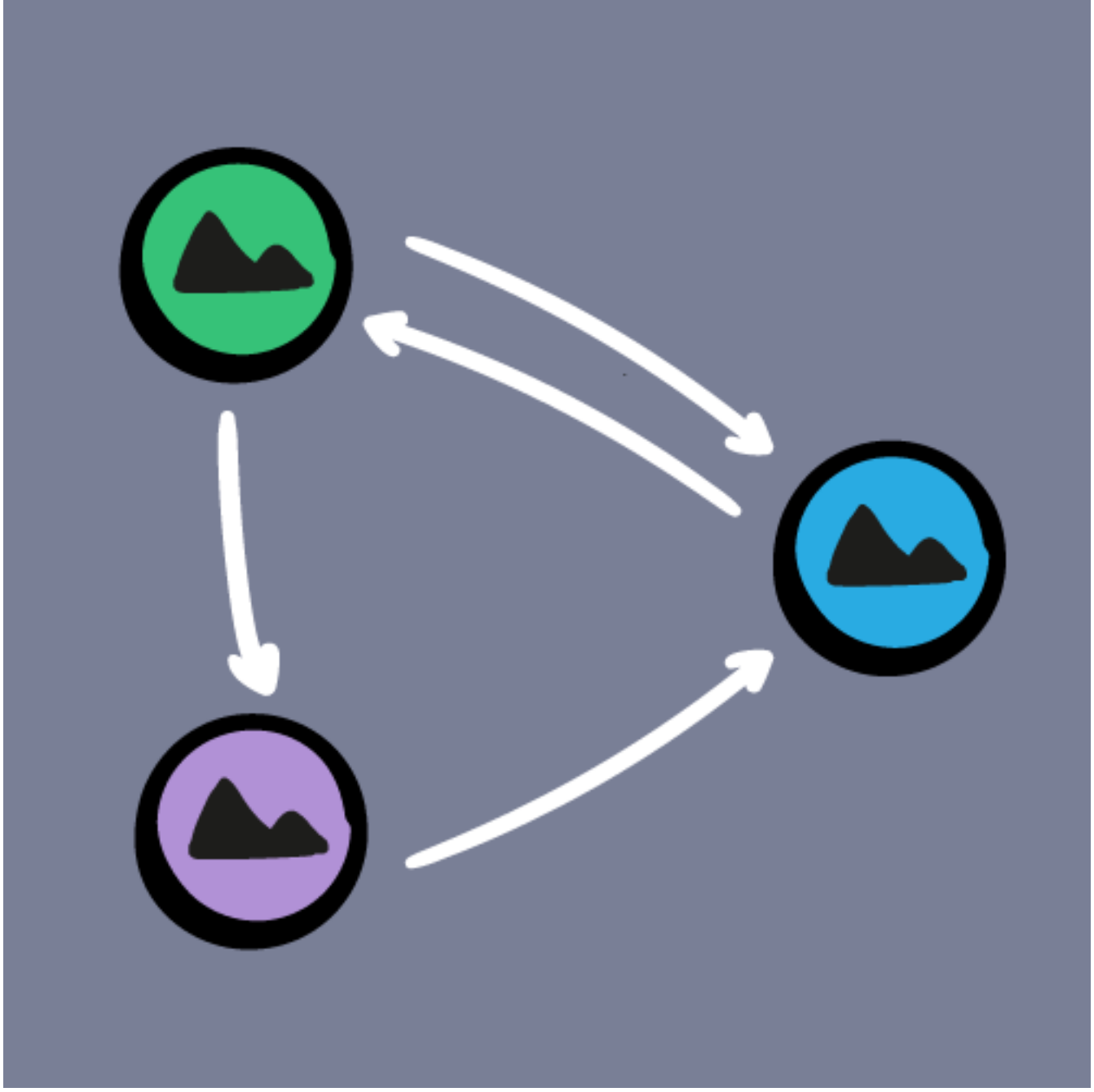
Operation and OperationQueue Tutorial in Swift

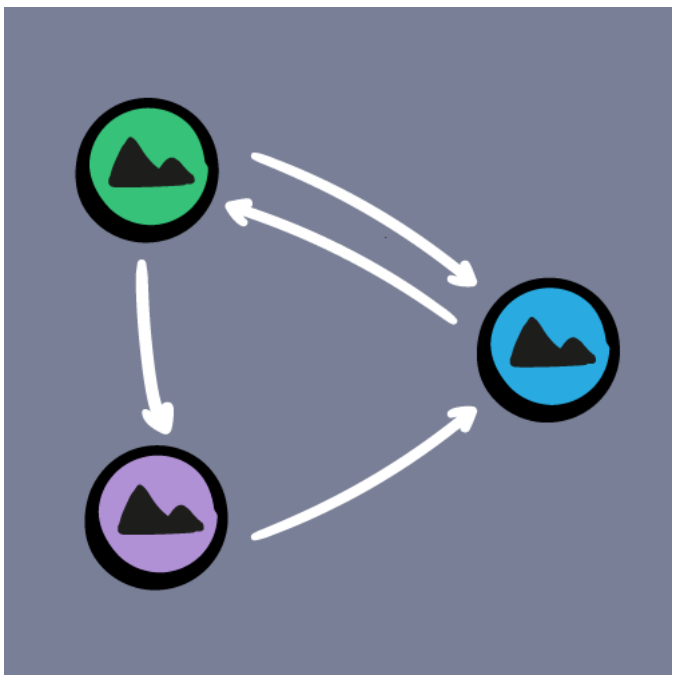
In this tutorial, you will create an app that uses concurrent operations to provide a responsive interface for users by using Operation and OperationQueue.

By James Goodwill Jun 27 2018 · Article (35 mins) · Intermediate

4.5/5 17 Ratings · Leave a Rating



Update note: James Goodwill updated this tutorial for Xcode 10 and Swift 4.2. Soheil Azarpour wrote the original post and Richard Turton completed a previous update.



Learn how to use Operations in your app!

Everyone has had the frustrating experience of tapping a button or entering some text in an iOS or Mac app, when all of a sudden: WHAM!

The user interface stops responding.

On the Mac, your users get to stare at the colorful wheel rotating for a while until they can interact with the UI again. In an iOS app, users expect apps to respond immediately to their touches. Unresponsive apps feel clunky and slow, and usually receive bad reviews.

Keeping your app responsive is easier said than done. Once your app needs to perform more than a handful of tasks, things get complicated quickly. There isn't much time to perform heavy work in the main run loop and still provide a responsive UI.

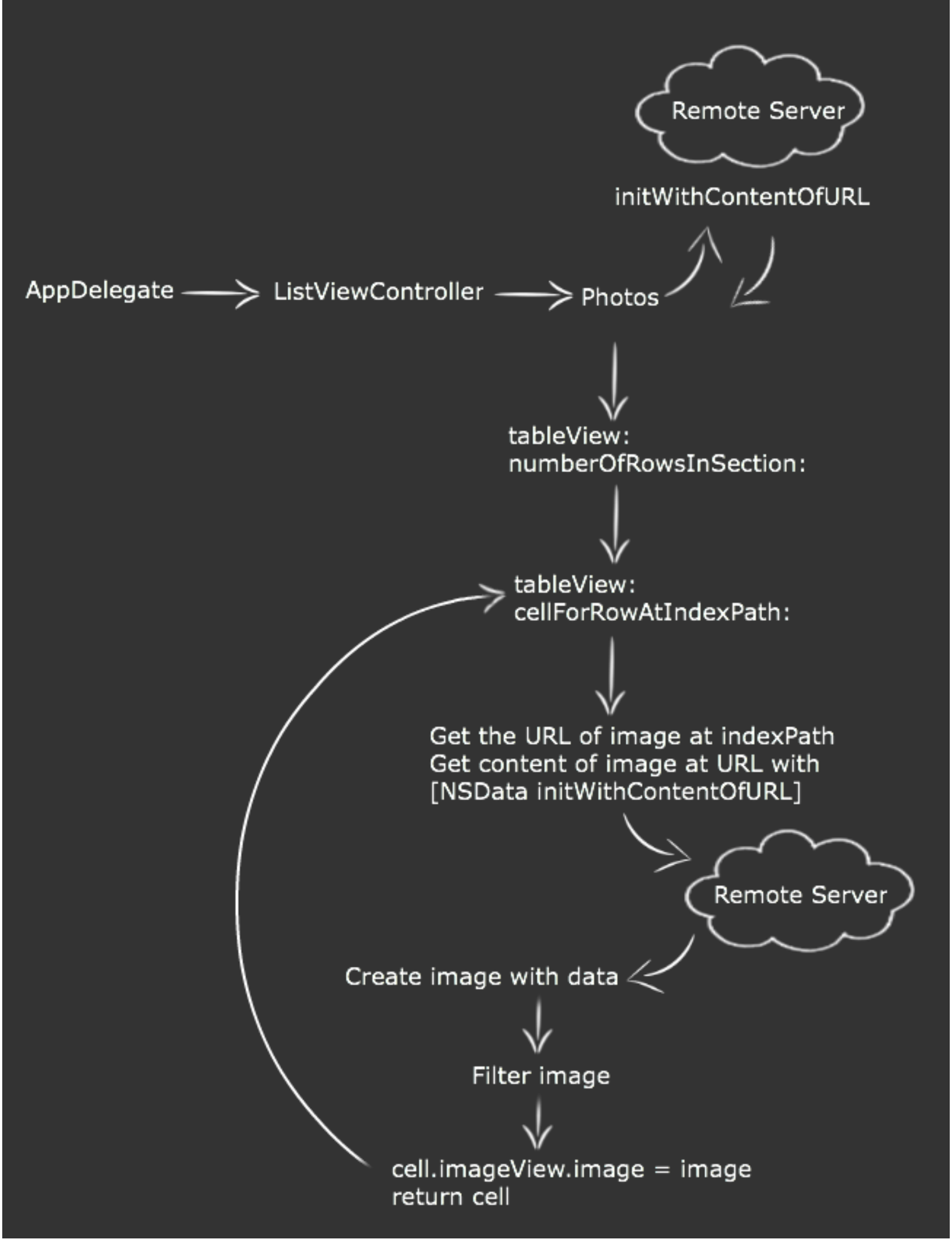
What's a poor developer to do? The solution is to move work off the main thread via concurrency. Concurrency means that your application executes multiple streams (or threads) of operations all at the same time. This way the user interface stays responsive as you're performing your work.

One way to perform operations concurrently in iOS is with the `Operation` and `OperationQueue` classes. In this tutorial, you'll learn how to use them! You'll start with an app that doesn't use concurrency at all, so it will appear very sluggish and unresponsive. Then, you'll rework the app to add concurrent operations and provide a more responsive interface to the user!

Getting Started

The overall goal of the sample project for this tutorial is to show a table view of filtered images. The images are downloaded from the Internet, have a filter applied, and then displayed in the table view.

Here's a schematic view of the app model:



Preliminary Model

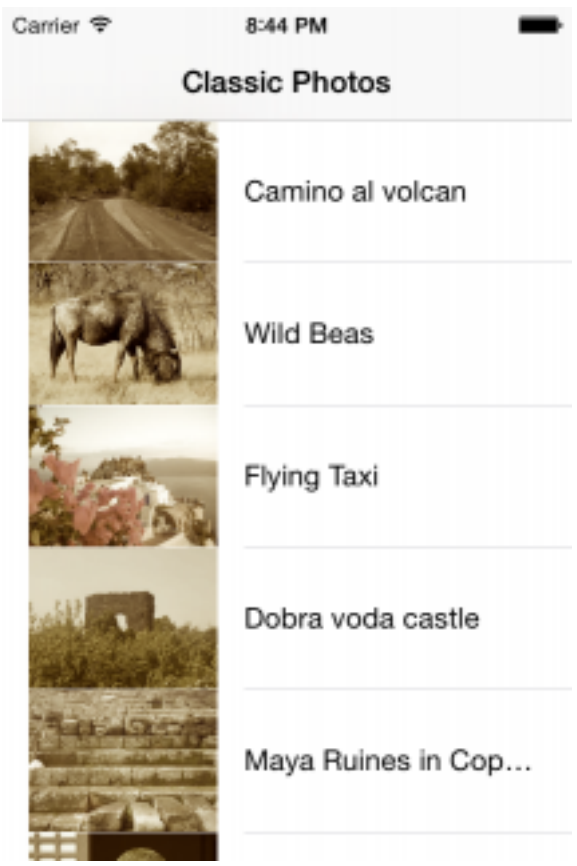
A First Try

Use the ***Download Materials*** button at the top or bottom of this tutorial

to download the starter project. It is the first version of the project that you'll be working on in this tutorial.

Note: All images are from stock.xchng. Some images in the data source are intentionally mis-named, so that there are instances where an image fails to download to exercise the failure case.

Build and run the project, and (eventually) you'll see the app running with a list of photos. Try scrolling the list. Painful, isn't it?



Classic photos, running slowly

All of the action is taking place in ***ListViewController.swift***, and most of *that* is inside `tableView(_:cellForRowAtIndexPath:)`.

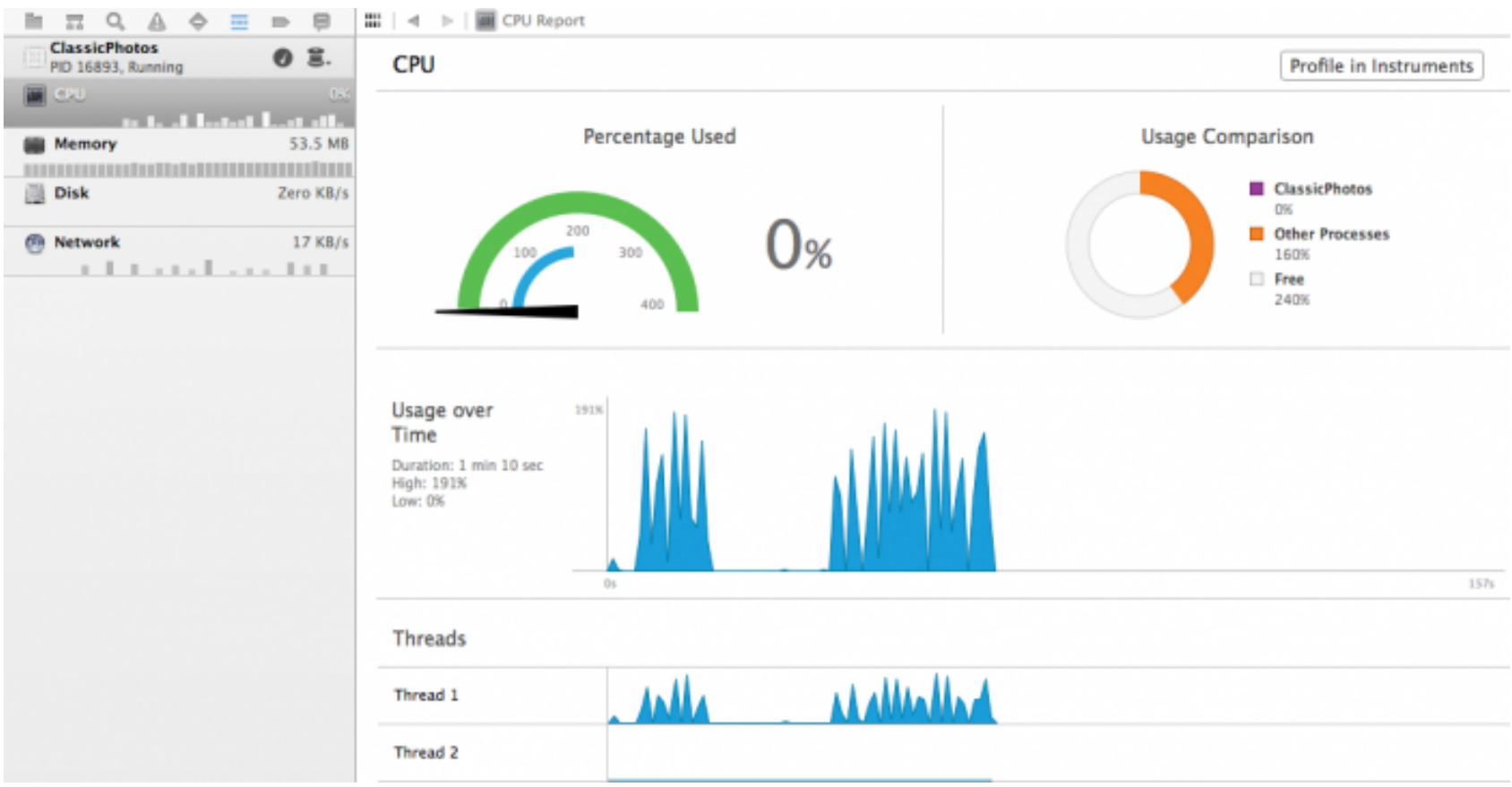
Have a look at that method and note there are two things taking place that are quite intensive:

1. ***Loading the image data from the web***. Even if this is easy work, the app still has to wait for the download to be complete before it can continue.
2. ***Filtering the image using Core Image***. This method applies a sepia filter to the image. If you would like to know more about Core Image filters, check out Beginning Core Image in Swift.

In addition, you're also loading the list of photos from the web when it is first requested:

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

All of this work is taking place on the main thread of the application. Since the main thread is also responsible for user interaction, keeping it busy with loading things from the web and filtering images is killing the responsiveness of the app. You can get a quick overview of this by using Xcode's gauges view. You can get to the gauges view by showing the ***Debug navigator*** (Command-7) and then selecting ***CPU*** while the app is running.



Xcode's Gauges view, showing heavy lifting on the main thread

You can see all those spikes in Thread 1, which is the main thread of the app. For more detailed information, you can run the app in Instruments, but that's a whole other tutorial. :]

It's time to think about how can you improve that user experience!

Tasks, Threads and Processes

Before going further, there are a few technical concepts you need to

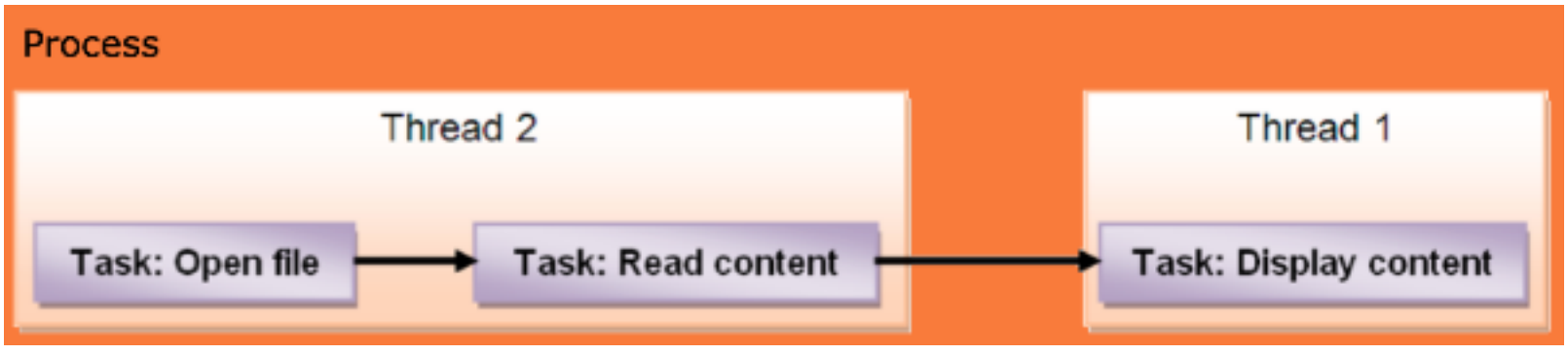
understand. Here are some key terms:

- ***Task***: a simple, single piece of work that needs to be done.
- ***Thread***: a mechanism provided by the operating system that allows multiple sets of instructions to operate at the same time within a single application.
- ***Process***: an executable chunk of code, which can be made up of multiple threads.

Note: In iOS and macOS, the threading functionality is provided by the POSIX Threads API (or pthreads) and is part of the operating system. This is pretty low level stuff, and you'll find that it's easy to make mistakes; perhaps the worst thing about threads is those mistakes can be incredibly hard to find!

The Foundation framework contains a class called ***Thread***, which is much easier to deal with, but managing multiple threads with Thread is still a headache. Operation and OperationQueue are higher level classes that have greatly simplified the process of dealing with multiple threads.

In this diagram, you can see the relationship between a process, threads, and tasks:



Process, Thread and Task

As you can see, a process can contain multiple threads of execution, and each thread can perform multiple tasks one at a time.

In this diagram, thread 2 performs the work of reading a file, while thread 1 performs user-interface related code. This is quite similar to how you should structure your code in iOS — the main thread performs any work related to the user interface, and secondary threads perform slow or

long-running operations such as reading files, accessing the network, etc.

Operation vs. Grand Central Dispatch (GCD)

You may have heard of Grand Central Dispatch (GCD). In a nutshell, GCD consists of language features, runtime libraries, and system enhancements to provide systemic and comprehensive improvements to support concurrency on multi-core hardware in iOS and macOS. If you'd like to learn more about GCD, you can read our Grand Central Dispatch Tutorial.

`Operation` and `OperationQueue` are built on top of GCD. As a very general rule, Apple recommends using the highest-level abstraction, then dropping down to lower levels when measurements show this is necessary.

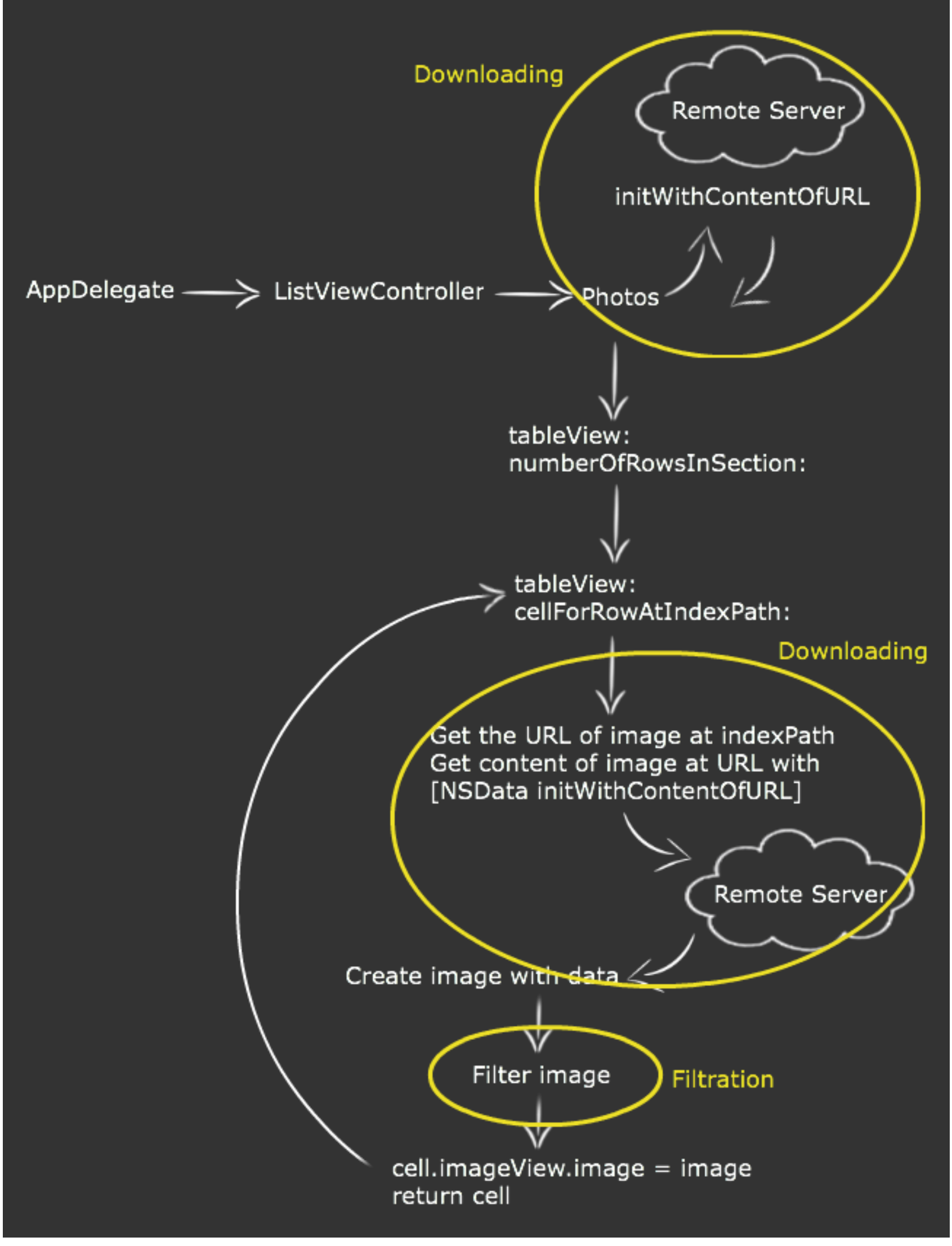
Here's a quick comparison of the two that will help you decide when and where to use GCD or `Operation`:

- ***GCD*** is a lightweight way to represent units of work that are going to be executed concurrently. You don't schedule these units of work; the system takes care of scheduling for you. Adding dependency among blocks can be a headache. Canceling or suspending a block creates extra work for you as a developer!
- ***Operation*** adds a little extra overhead compared to GCD, but you can add dependency among various operations and re-use, cancel or suspend them.

This tutorial will use `Operation` because you're dealing with a table view and, for performance and power consumption reasons, you need the ability to cancel an operation for a specific image if the user has scrolled that image off the screen. Even if the operations are on a background thread, if there are dozens of them waiting on the queue, performance will still suffer.

Refined App Model

It is time to refine the preliminary non-threaded model! If you take a closer look at the preliminary model, you'll see that there are three thread-bogging areas that can be improved. By separating these three areas and placing them in separate threads, the main thread will be relieved and can stay responsive to user interactions.



Improved model

To get rid of your application bottlenecks, you'll need a thread specifically to respond to user interactions, a thread dedicated to downloading data source and images, and a thread for performing image

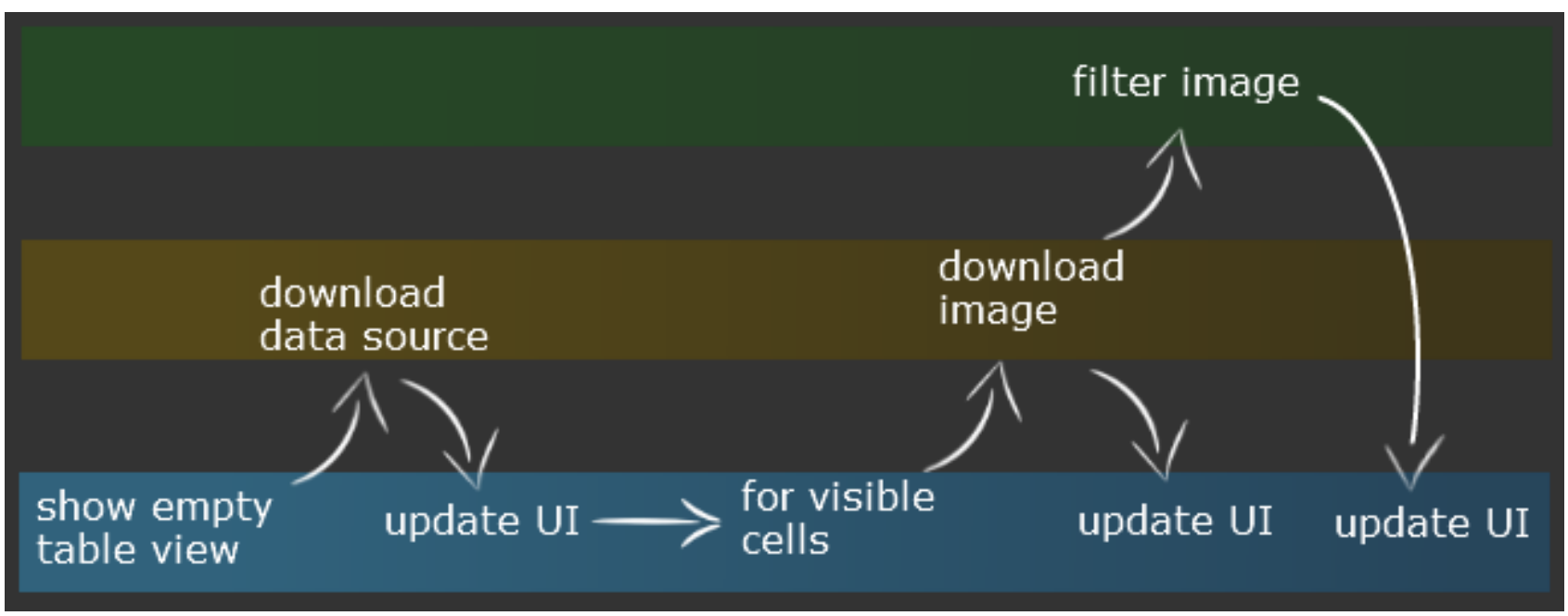
filtering. In the new model, the app starts on the main thread and loads an empty table view. At the same time, the app launches a second thread to download the data source.

Once the data source has been downloaded, you'll tell the table view to reload itself. This has to be done on the main thread, since it involves the user interface. At this point, the table view knows how many rows it has, and it knows the URL of the images it needs to display, but it doesn't have the actual images yet! If you immediately started to download all the images at this point, it would be terribly inefficient since you don't need all the images at once!

What can be done to make this better?

A better model is just to start downloading the images whose respective rows are visible on the screen. So your code will first ask the table view which rows are visible and, only then, will it start the download tasks. Similarly, the image filtering tasks can't begin until the image is completely downloaded. Therefore, the app shouldn't start the image filtering tasks until there is an unfiltered image waiting to be processed.

To make the app appear more responsive, the code will display the image right away once it is downloaded. It will then kick off the image filtering, then update the UI to display the filtered image. The diagram below shows the schematic control flow for this:



Control Flow

To achieve these objectives, you'll need to track whether the image is downloading, has downloaded, or is being filtered. You'll also need to track the status and type of each operation, so that you can cancel, pause or resume each as the user scrolls.

Okay! Now you're ready to get coding!

In Xcode, add a new ***Swift File*** to your project named ***PhotoOperations.swift***. Add the following code:

```
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
  }
}
```

This simple class represents each photo displayed in the app, together with its current state, which defaults to `.new`. The image defaults to a placeholder.

To track the status of each operation, you'll need a separate class. Add the following definition to the end of ***PhotoOperations.swift***:

```
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
  }()
}
```

This class contains two dictionaries to keep track of active and pending download and filter operations for each row in the table, and an operation queues for each type of operation.

All of the values are created lazily — they aren't initialized until they're first accessed. This improves the performance of your app.

Creating an `OperationQueue` is very straightforward, as you can see. Naming your queues helps with debugging, since the names show up in Instruments or the debugger. The `maxConcurrentOperationCount` is set to `1` for the sake of this tutorial to allow you to see operations finishing one by one. You could leave this part out and allow the queue to decide how many operations it can handle at once — this would further improve performance.

How does the queue decide how many operations it can run at once? That's a good question! It depends on the hardware. By default, `OperationQueue` does some calculation behind the scenes, decides what's best for the particular platform it's running on, and launches the maximum possible number of threads.

Consider the following example: Assume the system is idle and there are lots of resources available. In this case, the queue may launch eight simultaneous threads. Next time you run the program, the system may be busy with other, unrelated operations which are consuming resources. This time, the queue may launch only two simultaneous threads. Because

you've set a maximum concurrent operations count in this app, only one operation will happen at a time.

Note: You might wonder why you have to keep track of all active and pending operations. The queue has an `operations` method which returns an array of operations, so why not use that? In this project, it won't be very efficient to do so. You need to track which operations are associated with which table view rows, which would involve iterating over the array each time you needed one. Storing them in a dictionary with the index path as a key means lookup is fast and efficient.

It's time to take care of download and filtration operations. Add the following code to the end of ***PhotoOperations.swift***:

```
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")
    }
  }
}
```

`Operation` is an abstract class, designed for subclassing. Each subclass represents a specific ***task*** as represented in the diagram earlier.

Here's what's happening at each of the numbered comments in the code above:

1. Add a constant reference to the `PhotoRecord` object related to the operation.
2. Create a designated initializer allowing the photo record to be passed in.
3. `main()` is the method you override in `Operation` subclasses to actually perform work.
4. Check for cancellation before starting. Operations should regularly check if they have been cancelled before attempting long or intensive work.
5. Download the image data.
6. Check again for cancellation.
7. If there is data, create an image object and add it to the record, and move the state along. If there is no data, mark the record as failed and set the appropriate image.

Next, you'll create another operation to take care of image filtering. Add the following code to the end of ***PhotoOperations.swift***:

```
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
    }
  }
}
```

This looks very similar to the downloading operation, except that you're applying a filter to the image (using an as yet unimplemented method, hence the compiler error) instead of downloading it.

Add the missing image filter method to the `ImageFiltration` class:

```
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)
}
```

The image filtering is the same implementation used previously in `ListViewController`. It's been moved here so that it can be done as a separate operation in the background. Again, you should check for cancellation very frequently; a good practice is to do it before and after any expensive method call. Once the filtering is done, you set the values of the photo record instance.

Great! Now you have all the tools and foundation you need in order to process operations as background tasks. It's time to go back to the view controller and modify it to take advantage of all these new benefits.

Switch to ***ListViewController.swift*** and delete the `lazy var photos` property declaration. Add the following declarations instead:

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

These properties hold an array of the `PhotoRecord` objects and a `PendingOperations` object to manage the operations.

Add a new method to the class to download the photos property list:

```
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()
}
```

Here's what this does:

1. Create a `URLSession` data task to download the property list of images on a background thread.
2. Configure a `UIAlertController` to use in the event of an error.
3. If the request succeeds, create a dictionary from the property list. The dictionary uses the image name as the key and its URL as the value.
4. Build the array of `PhotoRecord` objects from the dictionary.
5. Return to the main thread to reload the table view and display the images.
6. Display the alert controller in the event of an error. Remember that `URLSession` tasks run on background threads and display of any messages on the screen must be done from the main thread.
7. Run the download task.

Call the new method at the end of `viewDidLoad()`:

```
fetchPhotoDetails()
```

Next, find `tableView(_:cellForRowAtIndexPath:)` — it'll be easy to find because the compiler is complaining about it — and replace it with the following implementation:

```
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
}
```

Here's what this does:

1. To provide feedback to the user, create a `UIActivityIndicatorView` and set it as the cell's accessory view.
2. The data source contains instances of `PhotoRecord`. Fetch the correct one based on the current indexPath.
3. The cell's text label is (nearly) always the same and the image is set appropriately on the `PhotoRecord` as it is processed, so you can set them both here, regardless of the state of the record.
4. Inspect the record. Set up the activity indicator and text as appropriate, and kick off the operations (not yet implemented).

Add the following method to the class to start the operations:

```
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")
```

```
  }
}
```

Here, you pass in an instance of `PhotoRecord` along with its index path. Depending on the photo record's state, you kick off either a download or filter operation.

Note: The methods for downloading and filtering images are implemented separately, as there is a possibility that while an image is being downloaded the user could scroll away, and you won't yet have applied the image filter. So next time the user comes to the same row, you don't need to re-download the image; you only need to apply the image filter! Efficiency rocks! :]

Now you need to implement the methods that you called in the method above. Remember that you created a custom class, `PendingOperations`, to keep track of operations; now you actually get to use it! Add the following methods to the class:

```
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)
}
```

Here's a quick list to make sure you understand what's going on in the code above:

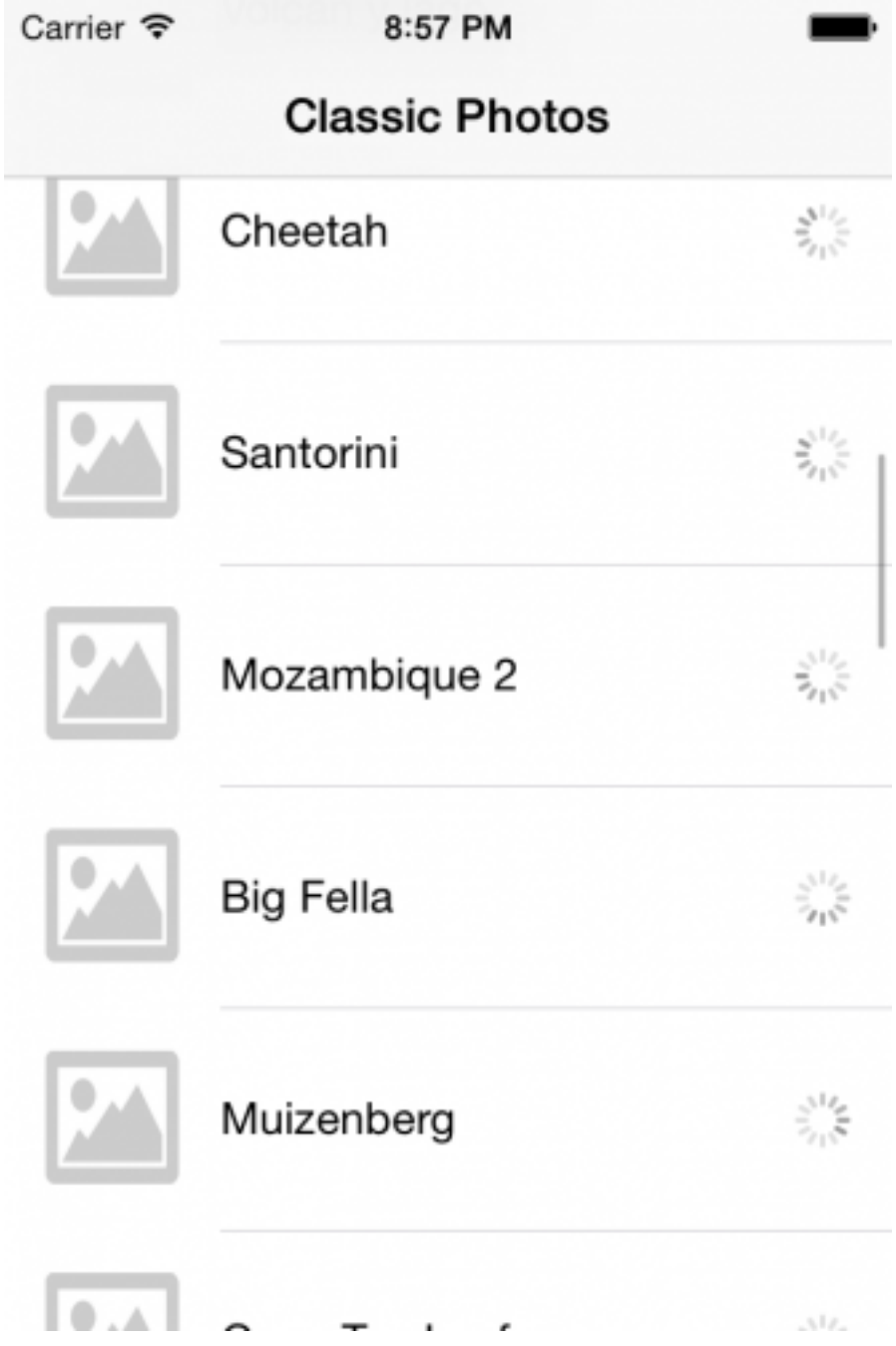
1. First, check for the particular `indexPath` to see if there is already an operation in `downloadsInProgress` for it. If so, ignore this request.
2. If not, create an instance of `ImageDownloader` by using the designated initializer.
3. Add a completion block which will be executed when the operation is completed. This is a great place to let the rest of your app know that an operation has finished. It's important to note that the completion block is executed even if the operation is cancelled, so you must check this property before doing anything. You also have no guarantee of which thread the completion block is called on, so you

need to use GCD to trigger a reload of the table view on the main thread.

4. Add the operation to `downloadsInProgress` to help keep track of things.
5. Add the operation to the download queue. This is how you actually get these operations to start running — the queue takes care of the scheduling for you once you've added the operation.

The method to filter the image follows the same pattern, except it uses `ImageFiltration` and `filtrationsInProgress` to track the operations. As an exercise, you could try getting rid of the repetition in this section of code :]

You made it! Your project is complete. Build and run to see your improvements in action! As you scroll through the table view, the app no longer stalls and starts downloading images and filtering them as they become visible.



Classic photos, now with actual scrolling!

Isn't that cool? You can see how a little effort can go a long way towards making your applications a lot more responsive — and a lot more fun for the user!

Fine Tuning

You've come a long way in this tutorial! Your little project is responsive and shows lots of improvement over the original version. However, there are still some small details that are left to take care of.

You may have noticed that as you scroll in the table view, those off-screen cells are still in the process of being downloaded and filtered. If you scroll quickly, the app will be busy downloading and filtering images from the cells further back in the list even though they aren't visible. Ideally, the app should cancel filtering of off-screen cells and prioritize the cells that are currently displayed.

Didn't you put cancellation provisions in your code? Yes, you did — now you should probably make use of them! :]

Open ***ListViewController.swift***. Go to the implementation of `tableView(_:cellForRowAtIndexPath:)`, and wrap the call to `startOperationsForPhotoRecord` in an `if` statement, as follows:

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

You tell the table view to start operations *only if the table view is not scrolling*. These are actually properties of `UIScrollView` and, because `UITableView` is a subclass of `UIScrollView`, table views automatically inherit these properties.

Next, add the implementation of the following `UIScrollView` delegate methods to the class:

```
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()
}
```

A quick walk-through of the code above shows the following:

1. As soon as the user starts scrolling, you will want to suspend all operations and take a look at what the user wants to see. You will implement `suspendAllOperations` in just a moment.
2. If the value of decelerate is `false`, that means the user stopped dragging the table view. Therefore you want to resume suspended operations, cancel operations for off-screen cells, and start operations for on-screen cells. You will implement `loadImagesForOnscreenCells` and `resumeAllOperations` in a little while, as well.
3. This delegate method tells you that table view stopped scrolling, so you will do the same as in #2.

Now, add the implementation of these missing methods to ***ListViewController.swift***:

```
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)
    }
  }
}
```

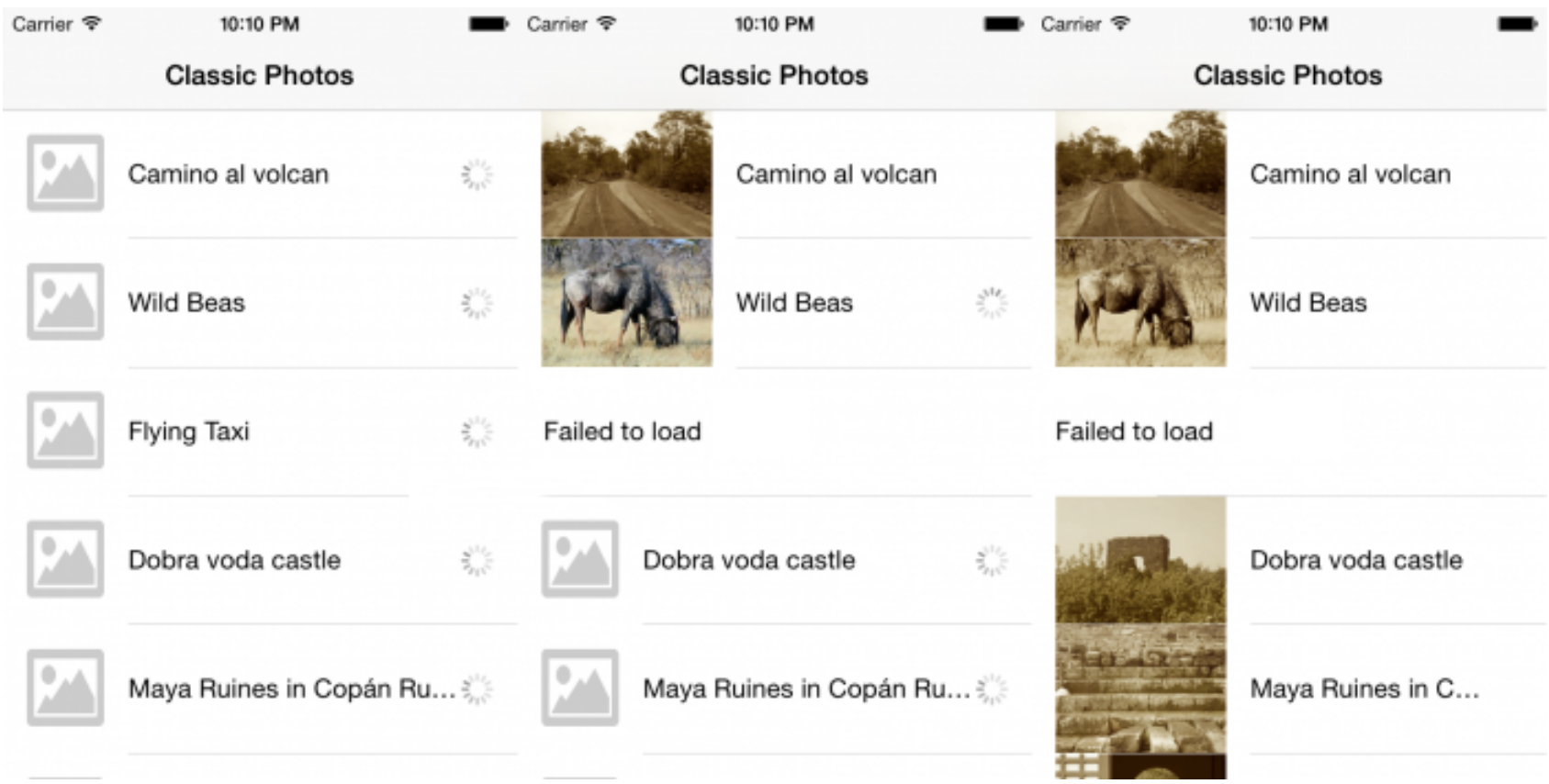
`suspendAllOperations()` and `resumeAllOperations()` have straightforward implementations. `OperationQueue`s can be suspended, by setting the `suspended` property to `true`. This will suspend all operations in a queue — you can't suspend operations individually.

`loadImagesForOnscreenCells()` is a little more complex. Here's what's going on:

1. Start with an array containing index paths of all the currently visible rows in the table view.
2. Construct a set of all pending operations by combining all the downloads in progress and all the filters in progress.
3. Construct a set of all index paths with operations to be cancelled. Start with all operations, and then remove the index paths of the visible rows. This will leave the set of operations involving off-screen rows.

4. Construct a set of index paths that need their operations started. Start with index paths all visible rows, and then remove the ones where operations are already pending.
5. Loop through those to be cancelled, cancel them, and remove their reference from `PendingOperations`.
6. Loop through those to be started, and call `startOperations(for:at:)` for each.

Build and run and you should have a more responsive and better resource-managed application! Give yourself a round of applause!



Classic photos, loading things one step at a time!

Notice that when you finish scrolling the table view, the images on the visible rows will start processing right away.

Where to Go From Here?

You can download the completed version of the project using the ***Download Materials*** button at the top or bottom of this tutorial.

You've learned how to use `Operation` and `OperationQueue` to move long-running computations off of the main thread while keeping your source code maintainable and easy to understand.

But beware — like deeply-nested blocks, gratuitous use of multi-threading can make a project incomprehensible to people who have to maintain your code. Threads can introduce subtle bugs that may never appear until your network is slow, or the code is run on a faster (or slower) device, or one with a different number of cores. Test very carefully and always use Instruments (or your own observations) to verify that introducing threads really has made an improvement.

A useful feature of operations that isn't covered here is dependency. You can make an operation dependent on one or more other operations. This operation then won't start until the operations on which it depends have all finished. For example:

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

filterOperation.addDependency(downloadOperation)
```

To remove dependencies:

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

Could the code in this project be simplified or improved by using dependencies? Put your new skills to use and try it. :] An important thing to note is that a dependent operation will still be started if the operations it depends on are *cancelled*, as well as if they finish naturally. You'll need to bear that in mind.

If you have any comments or questions about this tutorial or Operations in general, please join the forum discussion below!