



Home



My Network



Jobs



Messaging



Notifications



Me



For Business

[Get](#)
[It](#)

Multithreading - Smooth list scrolling in iOS

**Thomas Henry Oz Sandvik**Mobile App Architect | Senior iOS / MacOS / 8 articles
WatchOS Developer | Consultant | Softwa...[+ Follow](#)

June 13, 2023

If you've ever been in the midst of a technical interview or coding assignment, you might have encountered a scenario like the one I recently faced. The task was to load images asynchronously from a webservice into a UITableView. The catch was, I had to ensure that the app remained smooth and responsive, even when network speed was very slow, and I had to use reusable cells in the table view. It was a classic opportunity to apply multithreading in iOS, and also yet another free learning opportunity for me:-)

Juggling Tasks with Multithreading

Imagine this: your app is juggling several balls (tasks) at once. (haha, I know -- Trying to purposely make an analogy to throwing or juggling a ball) If it's a one-handed juggler (single-threaded), it can only handle one ball at a time. Toss in a ball that's a bit too heavy (eg. like fetching a large image from a server), or the network is bad, and your juggler gets stuck. Your app freezes up, users get frustrated, and balls start dropping everywhere.

Multithreading to the rescue! Now our app has several hands (threads), so it can juggle multiple balls. While one hand fetches that heavy image ball, the others can keep the rest of the balls (like user input, eg.) in the air. Even better,

when the network is slow, our many-handed app can still keep things moving along smoothly.

Managing Reusable Cells

UITableViews are fantastic, but they can be a bit tricky when it comes to loading and displaying images. Each cell that goes off-screen can be reused to display a new image. But here's where it gets tricky: *if the user scrolls quickly, and the image is still being fetched, the cell can be reused before the new image is ready*. You might end up showing the wrong images in your cells, which was happening to me... so the solution? We can use multithreading to cancel the image fetching when the cell is reused, and start fetching the new image. This way, the correct image always ends up displayed in the cell.

Using Async/Await in Swift in the Controller

```
/* Be aware: This code is very simple and is only for
illustrative purposes. In a real application, you should
add error handling and more.. */

class CustomTableViewController: UITableViewController {
    var imageURLs: [URL] = []

    override func tableView(_ tableView: UITableView,
                           cellForRowAt indexPath: IndexPath)
                           UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
            "Cell", for: indexPath)
        let imageURL = imageURLs[indexPath.row]

        Task {
            await cell.loadImage(from: imageURL)
        }

        return cell
    }
}

class CustomTableViewCell: UITableViewCell {
    @IBOutlet weak var imageView: UIImageView!

    override func prepareForReuse() {
        super.prepareForReuse()
        imageView.image = nil
    }

    func loadImage(from url: URL) async {
        do {
            let (data, _) = try await URLSession.shared.dataTask(
                completionHandler: { data, response, error in
                    if let image = UIImage(data: data) {
                        DispatchQueue.main.async {
                            self.imageView.image = image
                        }
                    }
                })
        } catch {
            print("Error loading image: \(error)")
        }
    }
}
```

```
    }
}
```

In the first example, I use an async method within ``loadImage(from:)`` to fetch the image asynchronously. I use ``Task {}`` to create a new asynchronous task when I call this method from ``cellForRowAt``. This task runs in the background, not blocking the UI thread. If the cell is reused before the image is fully fetched, ``prepareForReuse()`` will ensure the old image is removed, and a new fetch begins.

Using Async/Await in Swift in the Cell

```
class CustomTableViewCell: UITableViewCell
@IBOutlet weak var imageView: UIImageView!

var imageLoadingTask: Task.Handle<Never, Never>?

override func prepareForReuse() {
    super.prepareForReuse()

    imageLoadingTask?.cancel()
    imageLoadingTask = nil

    imageView.image = nil
}

func loadImage(from url: URL) {
    imageLoadingTask = async {
        do {
            let (data, _) = try await URLSession.shared.data(from: url)
            if let image = UIImage(data: data) {
                DispatchQueue.main.async {
                    self.imageView.image = image
                }
            }
        } catch {
            print("Error loading image: \(error)")
        }
    }
}

{
```

In this second example, I start a new asynchronous task each time ``loadImage(from:)`` is called, and save a reference to this task in the ``imageLoadingTask`` variable. If the cell is reused before the image is fully fetched, I cancel this task in ``prepareForReuse()``. This ensures that I do not waste resources loading images that are no longer needed.

This last one is the one that I ended up using, as it abstracted the async/await syntax away from my controller

The End

In the end, the coding assignment went well. I successfully applied multithreading to fetch images asynchronously and display them in reusable cells. So, when the network speed dropped, my app didn't miss a beat. Thanks to multithreading and the async/await syntax, I am able to present a smoothly functioning app and explain my approach in a clear and confident manner. This real-world problem underscores the importance of understanding and applying multithreading in iOS app development.

Report this

Published by



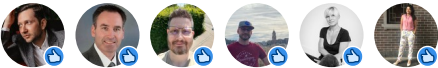
Thomas Henry Oz Sandvik
Mobile App Architect | Senior iOS / MacOS / WatchOS Developer | Consul...
Published • 1mo **8 articles**

+ Follow

In this article, I took on a coding challenge using Swift's multithreading to enable seamless image loading in a UITableView, regardless of network speed. The accomplishment emphasized the crucial role of multithreading in iOS.
[#AsyncAwait](#) [#Swift](#) [#Multithreading](#) [#ios](#)

Like Comment Share 7

Reactions



0 Comments




Add a comment...




Thomas Henry Oz Sandvik
Mobile App Architect | Senior iOS / MacOS / WatchOS Developer | Consultant
| Software Architect

+ Follow


More from Thomas Henry Oz Sandvik



To Freelance or Not To Freelance: A Developer's Guide to Navigate
Thomas Henry Oz Sandvik ...



Unpacking Redux and MVVM - A comparative look at the 2 architectures in...
Thomas Henry Oz Sandvik ...



Artificial Intelligence: The Biggest Threat to Knowledge Workers?

Thomas Henry Oz Sandvik ...

[See all 8 articles](#)