9. Loading UITableViewCell Images from an API

So far we’ve set up a Swift app that fetches and displays gists from the GitHub API. To do so it:

- Pulls public gists data from the GitHub gists API
- Turns the JSON response into an array of Swift structs
- Displays the results in a table view

In this chapter, we’ll add a new feature: displaying images of the gist owner’s avatar image in each row in the table view. We’ll get the URLs from the API then load the images asynchronously so that the UI isn’t held up while they’re loading. We’ll have to handle table view cells getting reused while we’re retrieving the images and we’ll set up an image cache so we don’t have to pull down the images every time a cell gets displayed.

If your API has images for your main items then follow along with this section. Otherwise, you might want to skip it and come back when you do need to load images from URLs, even if they’re not in a table view.

Here’s what it’ll look like when we’re done:

If you haven’t been following along, you might want to grab the code that’ll be our starting point from GitHub (tagged “tableview”)
1.

Or if you’d rather not type, grab the completed code from this chapter (tagged “cell_images”).

1https://github.com/cmoulton/grokSwiftREST_v1.4/releases/tag/tableview
2https://github.com/cmoulton/grokSwiftREST_v1.4/releases/tag/cell_images
9.1 Loading Images from URLs

We’ve already set up our Gist class to parse out the gist owner’s avatar URL using Codable.

```swift
struct Gist: Codable {
    struct Owner: Codable {
        var login: String
        var avatarURL: URL?

        enum CodingKeys: String, CodingKey {
            case login
            case avatarURL = "avatar_url"
        }
    }

    var id: String?
    var gistDescription: String?
    var url: URL?
    var owner: Owner?

    enum CodingKeys: String, CodingKey {
        case id
        case gistDescription = "description"
        case url
        case owner
    }
}
```

So we have everything set up to retrieve the image URLs. Now we need to get the actual image data from that URL. Adding to our GitHubAPIManager:

```swift
class GitHubAPIManager {
    // ...

    func imageFrom(url: URL,
                   completionHandler: @escaping (UIImage?, Error?) -> Void) {
        Alamofire.request(url)
            .responseData { response in
            guard let data = response.data else {
                completionHandler(nil, response.error)
                return
            }
            let image = UIImage(data: data)
```
That function will take the image URL and use it to make a GET request. When we get the results (as Data since we’re using .responseData) we check that there is data and if so try to turn it into an image. Then we pass that image back in the completion handler.

For this request we’re not using Codable because the response that we expect is just the data that represents the image. It’s not wrapped in JSON or any other format so we can just access the data directly by using response.data.

### 9.2 UITableViewCell Images from URLs

Now we can hook up displaying images. In MasterViewController’s tableView:(cellForRowAt indexPath:):

```swift
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
    let gist = gists[indexPath.row]
    cell.textLabel?.text = gist.description
    cell.detailTextLabel?.text = gist.owner?.login

    cell.imageView?.image = nil
    if let url = gist.owner?.avatarURL {
        GitHubAPIManager.shared.imageFrom(url: url) {
            (image, error) in
            guard error == nil else {
                print(error!)
                return
            }
            if let cellToUpdate = self.tableView?.cellForRow(at: indexPath) {
                cellToUpdate.imageView?.image = image // will work fine even if image is nil
                // need to reload the view, which won't happen otherwise
                // since this is in an async call
                cellToUpdate.setNeedsLayout()
            }
        }
    }
    return cell
}
```
We check that we have a URL for the image: if let url = gist.owner?.avatarURL. If so we fire off the method that we just wrote. Before we make that call, we set the image to nil in case the cell is being reused. We don’t want the image from another gist showing up until we get a response to the network request:

```swift
cell.imageView?.image = nil
if let url = gist.owner?.avatarURL {
    GitHubAPIManager.shared.imageFrom(url: url) {
        // ...
    }
}
```

In the completion handler we first check for errors:

```swift
GitHubAPIManager.shared.imageFrom(url: url) {
    (image, error) in
    guard error == nil else {
        print(error!)
        return
    }
    if let cellToUpdate = self.tableView?.cellForRow(at: indexPath) {
        cellToUpdate.imageView?.image = image // will work fine even if image is nil
        // need to reload the view, which won’t happen otherwise
        // since this is in an async call
        cellToUpdate.setNeedsLayout()
    }
}
```

If we don’t have an error we set the image that we received as the cell’s image. This is a bit complicated by the fact that our `imageFrom(url:)` function is asynchronous and by how table views reuse cells. Since we’re using `dequeueReusableCell(withIdentifier: for: indexPath)` the table view will reuse cells that have scrolled off of the screen. For example, if we have 20 gists but can only see 10 cells on the screen at once the table view will only create about 12 or 14 cells. If our cell has scrolled off of the screen then we shouldn’t set the image since that cell might end up being used to display a different gist.

We can work around this problem by using the index path to get the correct cell, if one exists, once we have the image:

```swift
if let cellToUpdate = self.tableView?.cellForRow(at: indexPath)
```

If the cell is currently being shown, that’ll give us the cell that’s on the screen for our index path. Then we can set the image on that cell. Since we’re all async here we also need to tell the cell that we’ve changed part of its view and it needs to redraw itself by using `cellToUpdate.setNeedsLayout()`. Otherwise the image wouldn’t get updated.
Note that we don’t need to check if our image is nil before setting it to the cell’s imageView. If the image is nil then `cellToUpdate.imageView?.image = image` will blank out the cell’s image.

Save and run. You should now see avatars for each gist as long as the user has set one:

If your API has images, set them up to load as we did in this section.

### 9.3 Enhancements

We’re firing off requests to get the images for each cell but sometimes by the time we get the result we don’t need it anymore. An optimization would be to cancel the network requests when the cell scrolls off of the screen. If you’re dealing with lots of images you’d want to do that (you’ll know you need it if the scrolling isn’t smooth). Our scrolling seems pretty smooth and we’re going to add a cache, so this concern isn’t one that needs to be addressed right now.

An optimization that is worthwhile even for this small app is caching the images so we don’t have to grab them from the web every time they’re shown. We’ll implement a quick and easy single-run-of-the-app cache to see how it can work. Then we’ll replace our cache with PINRemoteImage which will give us a smarter persistent cache.

### 9.4 Caching Images

In our `MasterViewController` we can add a dictionary to hold the images indexed by their absolute URL string:

```swift
var imageCache: [String: UIImage?] = []
```

Now we can save the images when we get them:
if let url = gist.owner?.avatarURL {
    GitHubAPIManager.shared.imageFrom(url: url) {
        (image, error) in
        guard error == nil else {
            print(error!)
            return
        }
        // Save the image so we won't have to keep fetching it if they scroll
        self.imageCache[url.absoluteString] = image
        if let cellToUpdate = self.tableView?.cellForRow(at: indexPath) {
            cellToUpdate.imageView?.image = image // will work fine even if image is nil
            // need to reload the view, which won't happen otherwise
            // since this is in an async call
            cellToUpdate.setNeedsLayout()
        }
    }
}

Before retrieving the image, we'll check the cache to see if we already have it:

override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
    let gist = gists[indexPath.row]
    cell.textLabel?.text = gist.gistDescription
    cell.detailTextLabel?.text = gist.owner?.login
    cell.imageView?.image = nil
    if let url = gist.owner?.avatarURL {
        if let cachedImage = imageCache[url.absoluteString] {
            cell.imageView?.image = cachedImage
        } else {
            GitHubAPIManager.shared.imageFrom(url: url) {
                (image, error) in
                // ...
            }
        }
    }
    return cell
}

So finally our `tableView(_:cellForRowAt:)` function will look like:
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
    let gist = gists[indexPath.row]
    cell.textLabel?.text = gist.gistDescription
    cell.detailTextLabel?.text = gist.owner?.login

    cell.imageView?.image = nil
    if let url = gist.owner?.avatarURL {
        if let cachedImage = imageCache[url.absoluteString] {
            cell.imageView?.image = cachedImage
        } else {
            GitHubAPIManager.shared.imageFrom(url: url) {
                image, error
            in
                guard error == nil else {
                    print(error)
                    return
                }

                // Save the image so we won’t have to keep fetching it if they scroll
                self.imageCache[url.absoluteString] = image
            }
        }
    }

    if let cellToUpdate = self.tableView?.cellForRow(at: indexPath) {
        cellToUpdate.imageView?.image = image // will work fine even if image is nil
        // need to reload the view, which won’t happen otherwise
        // since this is in an async call
        cellToUpdate.setNeedsLayout()
    }
}

return cell
}

To test out this code you’ll have to set a breakpoint and step through to see which lines get executed. To add a breakpoint, click on the line number next to the code where you want the code execution to stop:
Adding a breakpoint

Then run the app. When it gets to the breakpoint the code will stop running. At that point you can use the panel at the bottom of Xcode to examine variables and step through the code:

Click on the “step over” button to go to the next line of code:

Watch which code path gets taken by seeing which line is highlighted. If there is an `avatarURL` it will try to load it:
Load image

To resume the program until the next time the breakpoint is hit, click the continue button:

Continue button

When you run the app all of the images will get loaded from their URLs the first time that they’re displayed because we’re not persisting the image cache between runs of the app. We’ll fix that in the next section. Since table view cells get reused, our cache will kick in when we scroll cells on and off of the screen. The easiest way to see when the cache gets used is to move the breakpoint to where we’re loading an image from the cache:

Loading image from cache

After the initial cells load, scroll up and down. You should hit the breakpoint and see that line getting executed.

9.5 A Better Cache: PINRemoteImage

We’ve set up our API calls to parse an image URL out of JSON, then to load the image from the URL. We then used those images in UITableViewCells, handling the asynchronous loading even though the cells might have been reused. We set up a primitive cache for a single run of the app that helps
us avoid constantly reloading the images. Before using this code in a released app it would be worth setting up a persistent cache that’s used for multiple runs of the app.

**PINRemoteImage**³ is a persistent image caching library. Using this cache will make sure that images are only loaded once. It’s also simpler to integrate than our non-persistent cache.

Add PINRemoteImage v 2.1 to your project using **CocoaPods**. Import it into the MasterViewController file and remove our simple cache:

```swift
import UIKit
import PINRemoteImage

class MasterViewController: UITableViewController {

    var detailViewController: DetailViewController? = nil
    var gists = [Gist]()
    var imageCache: [String: UIImage?] = [:]

    // ...

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
        let gist = gists[indexPath.row]
        cell.textLabel?.text = gist.gistDescription
        cell.detailTextLabel?.text = gist.owner?.login

        cell.imageView?.image = nil
        if let url = gist.owner?.avatarURL {
            // ...
            return cell
        }

    // ...
    }

PINRemoteImage works best with a placeholder image. That’s also a nice visual touch. So open the **Assets.xcassets** asset library in your project (make sure you have the project organizer selected, it’s the first icon on the top-left and looks like a folder):

³https://github.com/pinterest/PINRemoteImage
Then rename an image to `placeholder.png` and drag & drop the image into the left panel under the `AppIcon placeholder`.

To use `PINRemoteImage` with a `UIImageView`, we call the `pin_setImage` function and pass in the URL, placeholder image, and a completion handler that is called after the image is set. Within the completion handler we can tell the cell it needs to refresh itself using `cellToUpdate.setNeedsLayout()`. We’ll display the placeholder for users who don’t have an avatar set as well as when we don’t have an image loaded yet:
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)

    let gist = gists[indexPath.row]
    cell.textLabel?.text = gist.description
    cell.detailTextLabel?.text = gist.owner?.login

    // set cell.imageView to display image at gist.owner?.avatarURL
    if let url = gist.owner?.avatarURL {
        cell.imageView?.pin_setImage(from: url, placeholderImage: UIImage(named: "placeholder.png")) {
            result in
            if let cellToUpdate = self.tableView?.cellForRow(at: indexPath) {
                cellToUpdate.setNeedsLayout()
            }
        }
    } else {
        cell.imageView?.image = UIImage(named: "placeholder.png")
    }

    return cell
}

9.6 And That’s All

Save and run to test that out. You should end up with some avatar images loaded and your placeholder image displayed for those who don’t have avatars:

If you got tired of typing, here’s the code with PINRemoteImage: (tagged “cell_images”)

⁴https://github.com/cmoulton/grokSwiftREST_v1.4/releases/tag/cell_images