Sample Code

Asynchronously loading images into table and collection views

Store and fetch images asynchronously to make your app more responsive.

Download

iOS 17.0+ | iPadOS 17.0+ | Mac Catalyst 17.0+ | Xcode 15.0+

Overview

Caching images can help you make the table and collection views in your app instantiate fast and respond quickly to scrolling. The app in the sample project demonstrates fetching images with URLs. The images are not part the assets catalog and instead are a part of the app bundle to simulate loading each asynchronously by URL. This ensures the user interface remains responsive. This project also supports Mac Catalyst.

Handle image loading and caching

In the sample, the class `ImageCache.swift` demonstrates a basic mechanism for image loading from a URL with `NSURLSession` and caching the downloaded images using `NSCache`. Views such as `UITableView` and `UICollectionView` are subclasses of `UIScrollView`.

As the user scrolls in a view, the app requests the same image repeatedly. This sample holds onto the relevant completion blocks until the image loads, then passes the image to all of the requesting blocks so the API only has to make one call to fetch an image for a given URL. The following code shows how the sample project consructs a basic caching and loading method:

```
// Returns the cached image if available, otherwise asynchronously loads and caches
final func load(url: NSURL, item: Item, completion: @escaping (Item, UIImage?) -> Sw
    // Check for a cached image.
```

```
    if let cachedImage = image(url: url) {
        DispatchQueue.main.async {
            completion(item, cachedImage)
        }
        return
    }
    // In case there are more than one requestor for the image, we append their comp
    if loadingResponses[url] != nil {
        loadingResponses[url]?.append(completion)
        return
    } else {
        loadingResponses[url] = [completion]
    }
    // Go fetch the image.
    ImageURLProtocol.urlSession().dataTask(with: url as URL) { (data, response, erro
        // Check for the error, then data and try to create the image.
        guard let responseData = data, let image = UIImage(data: responseData),
            let blocks = self.loadingResponses[url], error == nil else {
            DispatchQueue.main.async {
                completion(item, nil)
            }
            return
        }
        // Cache the image.
        self.cachedImages.setObject(image, forKey: url, cost: responseData.count)
        // Iterate over each requestor for the image and pass it back.
        for block in blocks {
            DispatchQueue.main.async {
                block(item, image)
            }
            return
        }
    }.resume()
}
```

Update your datasource responsibly

An app that loads all of its data on launch risks running out of memory or terminating for taking too long to complete. Unless the app requires all data to be loaded before operation, load your images as the UI requests them.

> **Note**
>
> To ensure items load before becoming visible on screen, make use of prefetching APIs when applicable. See Prefetching collection view data for best practices of prefetching data.

Generally the app should wait until the data source requests a cell to fetch and set an image. The sample project demonstrates one approach to fetching and displaying an image on a reusable view :

```
var content = cell.defaultContentConfiguration()
content.image = item.image
ImageCache.publicCache.load(url: item.url as NSURL, item: item) { (fetchedItem, imag
    if let img = image, img != fetchedItem.image {
        var updatedSnapshot = self.dataSource.snapshot()
        if let datasourceIndex = updatedSnapshot.indexOfItem(fetchedItem) {
            let item = self.imageObjects[datasourceIndex]
            item.image = img
            updatedSnapshot.reloadItems([item])
            self.dataSource.apply(updatedSnapshot, animatingDifferences: true)
        }
    }
}
cell.contentConfiguration = content
```

See Also

Data

Filling a table with data

Create and configure cells for your table dynamically using a data source object, or provide them statically from your storyboard.

`protocol` **UITableViewDataSource**

The methods that an object adopts to manage data and provide cells for a table view.

`protocol` **UITableViewDataSourcePrefetching**

A protocol that provides advance warning of the data requirements for a table view, allowing you to start potentially long-running data operations early.

`class` **UITableViewDiffableDataSource**

The object you use to manage data and provide cells for a table view.

`struct NSDiffableDataSourceSnapshot`

A representation of the state of the data in a view at a specific point in time.

`class UILocalizedIndexedCollation`

An object that organizes, sorts, and localizes the data for a table view that has a section index.

`protocol UIDataSourceTranslating`

An advanced interface for managing a data source object.

`class UIRefreshControl`

A standard control that can initiate the refreshing of a scroll view's contents.