Article

Downloading files from websites

Download files directly to the filesystem.

Overview

For network resources that are already stored as files, like images and documents, you can use download tasks to fetch these items directly to the local filesystem.

Tip

You can also configure download tasks to operate while your application is suspended or terminated in the background. See Downloading files in the background for details.

For simple downloads, use a completion handler

To download files, you create a `URLSessionDownloadTask` from a `URLSession`. If you don't care about receiving progress updates or other delegate callbacks during the download, you can use a completion handler. The task calls the completion handler when the download ends, either at the end of a successful download or when downloading fails.

Your completion handler may receive a client-side error, indicating a local problem like not being able to reach the network. If there is no client-side error, you also receive a `URLResponse`, which you should inspect to ensure that it indicates a successful response from the server.

If the download is successful, your completion handler receives a URL indicating the location of the downloaded file on the local filesystem. This storage is temporary. If you want to preserve the file, you *must* copy or move it from this location before returning from the completion handler.

The following example shows a simple example of creating a download task with a completion handler. If no errors are indicated, the completion handler moves the downloaded file to the app's `Documents` directory. Start the task by calling `resume()`.

Creating a download task with a completion handler

```
let downloadTask = URLSession.shared.downloadTask(with: url) {
    urlOrNil, responseOrNil, errorOrNil in
    // check for and handle errors:
    // * errorOrNil should be nil
    // * responseOrNil should be an HTTPURLResponse with statusCode in 200..<299

    guard let fileURL = urlOrNil else { return }
    do {
        let documentsURL = try
            FileManager.default.url(for: .documentDirectory,
                                    in: .userDomainMask,
                                    appropriateFor: nil,
                                    create: false)
        let savedURL = documentsURL.appendingPathComponent(fileURL.lastPathComponent
        try FileManager.default.moveItem(at: fileURL, to: savedURL)
    } catch {
        print ("file error: \(error)")
    }
}
downloadTask.resume()
```

Tip

The previous example creates the download task with `downloadTask(with:)`, which simply takes a `URL` parameter. If you need to customize the request you send to the server, create the task with `downloadTask(with:)` and pass in a customized `URLRequest`.

To receive progress updates, use a delegate

If you want to receive progress updates as the download proceeds, you must use a delegate. Instead of receiving the results in a completion handler, you receive callbacks to your implementations of methods from the `URLSessionTaskDelegate` and `URLSessionDownloadDelegate` protocols.

Create your own `URLSession` instance, and set its `delegate` property. The following example shows a lazily instantiated `urlSession` property that sets `self` as its delegate.

Creating a URL session with a delegate

```
private lazy var urlSession = URLSession(configuration: .default,
                                         delegate: self,
                                         delegateQueue: nil)
```

To start downloading, use this `URLSession` to create a `URLSessionDownloadTask`, and then start the task by calling `resume()`, as shown in in the following example.

Creating and starting a download task that uses a delegate

```
private func startDownload(url: URL) {
    let downloadTask = urlSession.downloadTask(with: url)
    downloadTask.resume()
    self.downloadTask = downloadTask
}
```

Receive progress updates

Once the download starts, you receive periodic progress updates in the `URLSessionDownloadDelegate` method `urlSession(_:downloadTask:didWriteData:totalBytesWritten:totalBytesExpectedToWrite:)`. You can use the byte counts provided by this callback to update a progress UI in your app.

The following example shows an implementation of this callback method. This implementation calculates the fractional progress of the download, and uses it to update a label that shows progress as a percentage. Because the callback is performed on an unknown Grand Central Dispatch queue, you *must* explicitly perform the UI update on the main queue.

Using a delegate method to update download progress in a UI

```
func urlSession(_ session: URLSession,
                downloadTask: URLSessionDownloadTask,
                didWriteData bytesWritten: Int64,
                totalBytesWritten: Int64,
                totalBytesExpectedToWrite: Int64) {
     if downloadTask == self.downloadTask {
        let calculatedProgress = Float(totalBytesWritten) / Float(totalBytesExpectec
        DispatchQueue.main.async {
            self.progressLabel.text = self.percentFormatter.string(from:
                NSNumber(value: calculatedProgress))
    }
}
```

Tip

If the only UI update you need to perform during the download is to update a UIProgressView, then use the task's progress property instead of performing your own progress calculations. This property is an instance of Progress that you can assign to the UIProgressView property observedProgress when you create the task to get automatic updating of the progress view.

Handle download completion or errors in your delegate

When you use a delegate instead of a completion handler, you handle the completion of the download by implementing `urlSession(_:downloadTask:didFinishDownloadingTo:)`. Check the `downloadTask`'s `response` property to ensure that the server response indicates success. If so, the `location` parameter provides a local URL where the file has been stored. This location is valid only until the end of the callback. This means you *must* either read the file immediately, or move it to another location such as the app's `Documents` directory before you return from the callback method. The following example shows how to preserve the downloaded file.

Saving the downloaded file in the delegate callback

```
func urlSession(_ session: URLSession,
                downloadTask: URLSessionDownloadTask,
                didFinishDownloadingTo location: URL) {
    // check for and handle errors:
    // * downloadTask.response should be an HTTPURLResponse with statusCode in 200..

    do {
        let documentsURL = try
            FileManager.default.url(for: .documentDirectory,
                                    in: .userDomainMask,
                                    appropriateFor: nil,
                                    create: false)
        let savedURL = documentsURL.appendingPathComponent(
            location.lastPathComponent)
        try FileManager.default.moveItem(at: location, to: savedURL)
    } catch {
        // handle filesystem error
    }
}
```

If a client-side error occurs, your delegate receives it in a callback to the `urlSession(_:task:didCompleteWithError:)` delegate method. On the other hand, if the download completes successfully, this method is called after `urlSession(_:downloadTask:didFinishDownloadingTo:)` and the error is `nil`.

See Also

Downloading

Pausing and resuming downloads

Allow the user to resume a download without starting over.

Downloading files in the background

Create tasks that download files while your app is inactive.