URLSession Tutorial: Getting Started

Audrey Tam on June 19, 2017

Update June 11, 2017: Updated by Audrey Tam for Xcode 9 beta / Swift 4. Original post by Ken Toh.

Whether an app retrieves application data from a server, updates your social media status or downloads remote files to disk, it's the HTTP network requests living at the heart of mobile applications that make the magic happen. To help you with the numerous requirements for network requests, Apple provides **`URLSession`**, a complete networking API for uploading and downloading content via HTTP.

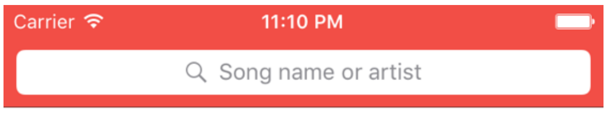
*Learn how to make HTTP data requests and implement file downloads with **`URLSession`**!*

In this **`URLSession`** tutorial, you'll learn how to build the ***Half Tunes*** app, which lets you query the iTunes Search API, then download 30-second previews of songs. The finished app will support *background transfers*, and let the user pause, resume or cancel in-progress downloads.

Getting Started

Download the starter project; it already contains a user interface to search for songs and display search results, networking service classes, and helper methods to store and play tracks. So you can focus on implementing the networking aspects of the app.

Build and run your project; you'll see a view with a search bar at the top and an empty table view below:

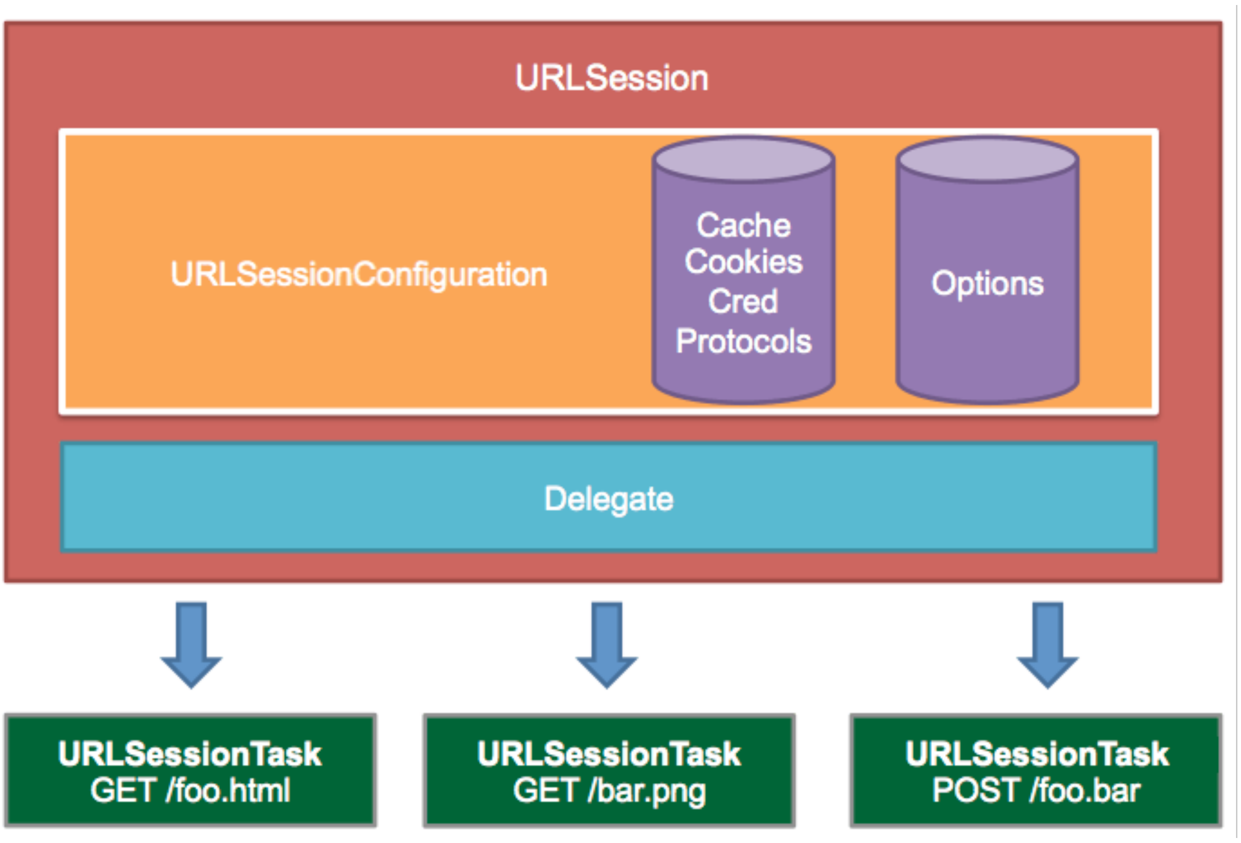


Type a query in the search bar, and tap ***Search***. The view remains empty, but don't worry: you'll change this with your new **`URLSession`** calls.

Overview of URLSession

Before you begin, it's important to appreciate **`URLSession`** and its constituent classes, so take a look at the quick overview below.

`URLSession` is technically *both* a class and a *suite of classes* for handling HTTP/HTTPS-based requests:



`URLSession` is the key object responsible for sending and receiving HTTP requests. You create it via **`URLSessionConfiguration`**, which comes in three flavors:

- **`.default`**: Creates a default configuration object that uses the disk-persisted global cache, credential and cookie storage objects.
- **`.ephemeral`**: Similar to the default configuration, except that all session-related data is stored in memory. Think of this as a

"private" session.

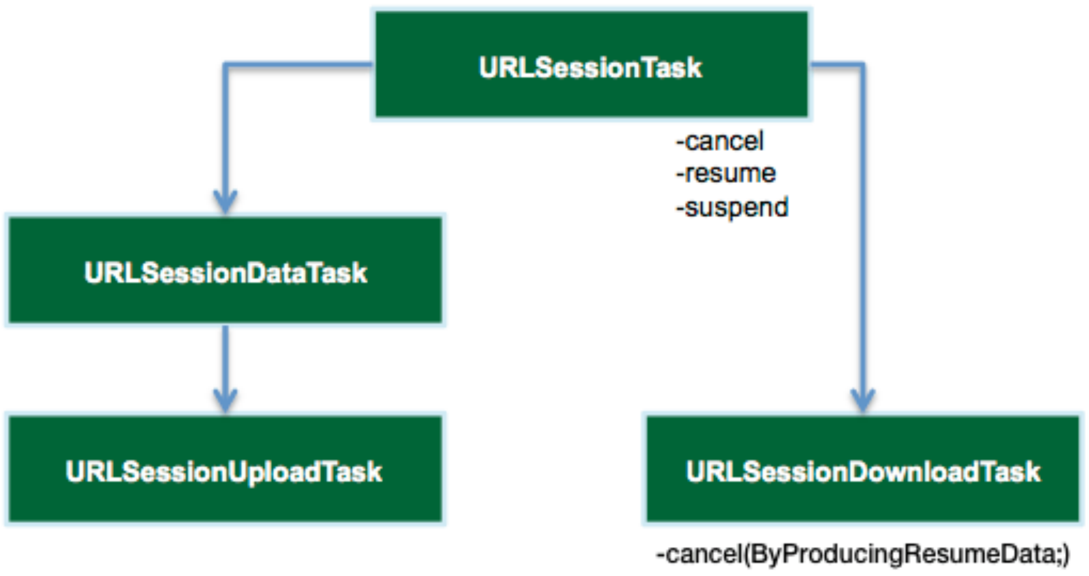
- **`.background`**: Lets the session perform upload or download tasks in the background. Transfers continue even when the app itself is suspended or terminated by the system.

`URLSessionConfiguration` also lets you configure session properties such as timeout values, caching policies and additional HTTP headers. Refer to the documentation for a full list of configuration options.

`URLSessionTask` is an abstract class that denotes a task object. A session creates one or more tasks to do the actual work of fetching data and downloading or uploading files.

There are three types of concrete session tasks:

- **`URLSessionDataTask`**: Use this task for HTTP GET requests to retrieve data from servers to memory.
- **`URLSessionUploadTask`**: Use this task to upload a file from disk to a web service, typically via a HTTP POST or PUT method.
- **`URLSessionDownloadTask`**: Use this task to download a file from a remote service to a temporary file location.



You can also suspend, resume and cancel tasks. **`URLSessionDownloadTask`** has the additional ability to pause for future resumption.

Generally, **`URLSession`** returns data in two ways: via a completion handler when a task finishes, either successfully or with an error, or by calling methods on a delegate that you set when creating the session.

Now that you have an overview of what **`URLSession`** can do, you're ready to put the theory into practice!

Data Task

You'll start by creating a data task to query the iTunes Search API for the user's search term.

In ***SearchVC+SearchBarDelegate.swift***, **`searchBarSearchButtonClicked(_:)`** first enables the network activity indicator on the status bar, to indicate to the user that a network process is running. Then it calls `getSearchResults(searchTerm:completion:)`, which is a stub in ***QueryService.swift***.

In ***Networking/QueryService.swift***, replace the first `// TODO` with the following:

```
// 1
let defaultSession = URLSession(configuration: .default)
// 2
var dataTask: URLSessionDataTask?
```

Here's what you've done:

1. You created a `URLSession`, and initialized it with a default session configuration.
2. You declared a `URLSessionDataTask` variable, which you'll use to make an HTTP GET request to the iTunes Search web service when the user performs a search. The data task will be re-initialized each time the user enters a new search string.

Next, replace the `getSearchResults(searchTerm:completion:)` stub with the following:

```
func getSearchResults(searchTerm: String, completion: @escaping QueryR
  // 1
  dataTask?.cancel()
  // 2
  if var urlComponents = URLComponents(string: "https://itunes.apple.c
    urlComponents.query = "media=music&entity=song&term=\(searchTerm)"
    // 3
    guard let url = urlComponents.url else { return }
    // 4
    dataTask = defaultSession.dataTask(with: url) { data, response, er
      defer { self.dataTask = nil }
      // 5
      if let error = error {
        self.errorMessage += "DataTask error: " + error.localizedDescr
      } else if let data = data,
        let response = response as? HTTPURLResponse,
        response.statusCode == 200 {
        self.updateSearchResults(data)
        // 6
        DispatchQueue.main.async {
          completion(self.tracks, self.errorMessage)
        }
      }
    }
    // 7
    dataTask?.resume()
  }
}
```

Taking each numbered comment in turn:

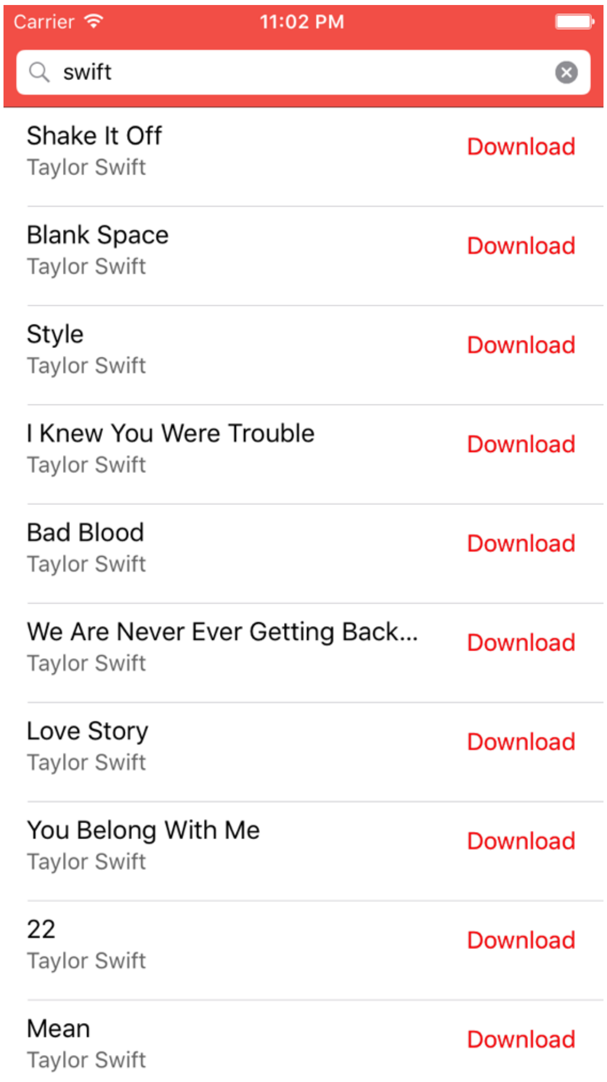
1. For a new user query, you cancel the data task if it already exists, because you want to reuse the data task object for this new query.
2. To include the user's search string in the query URL, you create a `URLComponents` object from the iTunes Search base URL, then set its query string: this ensures that characters in the search string are properly escaped.
3. The `url` property of `urlComponents` might be nil, so you optional-bind it to `url`.
4. From the session you created, you initialize a `URLSessionDataTask` with the query `url` and a completion handler to call when the data task completes.
5. If the HTTP request is successful, you call the helper method `updateSearchResults(_:)`, which parses the response `data` into the `tracks` array.
6. You switch to the main queue to pass `tracks` to the completion handler in ***SearchVC+SearchBarDelegate.swift***.

7. All tasks start in a suspended state by default; calling **`resume()`** starts the data task.

Now flip back to the **`getSearchResults(searchTerm:completion:)`** completion handler in ***SearchVC+SearchBarDelegate.swift***: after hiding the activity indicator, it stores **`results`** in **`searchResults`**, then updates the table view.

Note: The default request method is GET. If you want a data task to POST, PUT or DELETE, create a **`URLRequest`** with the **`url`**, set the request's **`HTTPMethod`** property appropriately, then create a data task with the **`URLRequest`**, instead of with the **`URL`**.

Build and run your app; search for any song and you'll see the table view populate with the relevant track results like so:



With a bit of **`URLSession`** magic added, Half Tunes is now a bit functional!

Download Task

Being able to view song results is nice, but wouldn't it be better if you could tap on a song to download it? That's precisely your next order of business. You'll use a download task, which makes it easy to save the song snippet in a local file.

Download Class

To make it easy to handle multiple downloads, you'll first create a custom object to hold the state of an active download.

Create a new Swift file named ***Download.swift*** in the ***Model*** group.

Open ***Download.swift***, and add the following implementation:

```
class Download {

  var track: Track
  init(track: Track) {
    self.track = track
  }

  // Download service sets these values:
  var task: URLSessionDownloadTask?
  var isDownloading = false
  var resumeData: Data?

  // Download delegate sets this value:
  var progress: Float = 0

}
```

Here's a rundown of the properties of `Download`:

- ***track***: The track to download. The track's `url` property also acts as a unique identifier for a `Download`.
- ***task***: The `URLSessionDownloadTask` that downloads the track.
- ***isDownloading***: Whether the download is ongoing or paused.
- ***resumeData***: Stores the `Data` produced when the user pauses a download task. If the host server supports it, your app can use this to resume a paused download in the future.
- ***progress***: The fractional progress of the download: a float between 0.0 and 1.0.

Next, in ***Networking/DownloadService.swift***, add the following property at the top of the class:

```
var activeDownloads: [URL: Download] = [:]
```

This dictionary simply maintains a mapping between a URL and its active **`Download`**, if any.

URLSessionDownloadDelegate

You *could* create your download task with a completion handler, like the data task you just created. But later in this tutorial, you'll monitor and update the download progress: for that, you'll need to implement a custom delegate, so you might as well do that now.

There are several session delegate protocols, listed in [URLSession documentation](). **`URLSessionDownloadDelegate`** handles task-level events specific to download tasks.

You'll soon set **`SearchViewController`** as the session delegate, so first create an extension to conform to the session delegate protocol.

Create a new Swift file named ***SearchVC+URLSessionDelegates.swift*** in the ***Controller*** group. Open it, and create the following **`URLSessionDownloadDelegate`** extension:

```
extension SearchViewController: URLSessionDownloadDelegate {
  func urlSession(_ session: URLSession, downloadTask: URLSessionDownl
    didFinishDownloadingTo location: URL) {
    print("Finished downloading to \(location).")
  }
}
```

The only non-optional **`URLSessionDownloadDelegate`** method is

`urlSession(_:downloadTask:didFinishDownloadingTo:)`, which is called when a download finishes. For now, you'll just print a message whenever a download completes.

Creating a Download Task

With all the preparatory work out of the way, you're now ready to implement file downloads. You'll first create a dedicated session to handle your download tasks.

In ***Controller/SearchViewController.swift***, add the following code right before **`viewDidLoad()`**:

```
lazy var downloadsSession: URLSession = {
  let configuration = URLSessionConfiguration.default
  return URLSession(configuration: configuration, delegate: self, dele
}()
```

Here you initialize a separate session with a default configuration, and specify a delegate, which lets you receive **`URLSession`** events via delegate calls. This will be useful for monitoring the progress of the task.

Setting the delegate queue to **`nil`** causes the session to create a serial operation queue to perform all calls to delegate methods and completion handlers.

Note the lazy creation of **`downloadsSession`**: this lets you delay the creation of the session until *after* the view controller is initialized, which allows you to pass **`self`** as the delegate parameter to the session initializer.

Now add this line at the end of **`viewDidLoad()`**:

```
downloadService.downloadsSession = downloadsSession
```

This sets the **`downloadsSession`** property of **`DownloadService`**.

With your session and delegate configured, you're finally ready to create a download task when the user requests a track download.

In ***Networking/DownloadService.swift***, replace the **`startDownload(_:)`** stub with the following implementation:

```
func startDownload(_ track: Track) {
  // 1
  let download = Download(track: track)
  // 2
  download.task = downloadsSession.downloadTask(with: track.previewURL
  // 3
  download.task!.resume()
  // 4
  download.isDownloading = true
  // 5
  activeDownloads[download.track.previewURL] = download
}
```

When the user taps a table view cell's ***Download*** button, **`SearchViewController`**, acting as **`TrackCellDelegate`**, identifies the **`Track`** for this cell, then calls **`startDownload(_:)`** with this **`Track`**. Here's what's going on in **`startDownload(_:)`**:

1. You first initialize a **`Download`** with the track.
2. Using your new session object, you create a **`URLSessionDownloadTask`** with the track's preview URL, and set it to the **`task`** property of the **`Download`**.
3. You start the download task by calling **`resume()`** on it.
4. You indicate that the download is in progress.

5. Finally, you map the download URL to its **`Download`** in the **`activeDownloads`** dictionary.

Build and run your app; search for any track and tap the ***Download*** button on a cell. After a while, you'll see a message in the debug console signifying that the download is complete. The Download button remains, but you'll fix that soon. First, you want to play some tunes!

Saving and Playing the Track

When a download task completes, **`urlSession(_:downloadTask:didFinishDownloadingTo:)`** provides a URL to the temporary file location: you saw this in the print message. Your job is to move it to a permanent location in your app's sandbox container directory before you return from the method.

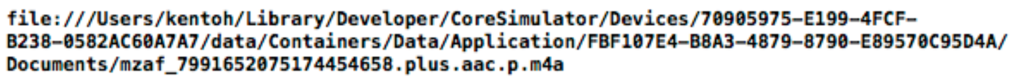
In ***SearchVC+URLSessionDelegates***, replace the print statement in **`urlSession(_:downloadTask:didFinishDownloadingTo:)`** with the following code:

```
// 1
guard let sourceURL = downloadTask.originalRequest?.url else { return
let download = downloadService.activeDownloads[sourceURL]
downloadService.activeDownloads[sourceURL] = nil
// 2
let destinationURL = localFilePath(for: sourceURL)
print(destinationURL)
// 3
let fileManager = FileManager.default
try? fileManager.removeItem(at: destinationURL)
do {
  try fileManager.copyItem(at: location, to: destinationURL)
  download?.track.downloaded = true
} catch let error {
  print("Could not copy file to disk: \(error.localizedDescription)")
}
// 4
if let index = download?.track.index {
  DispatchQueue.main.async {
    self.tableView.reloadRows(at: [IndexPath(row: index, section: 0)],
  }
}
```

Here's what you're doing at each step:

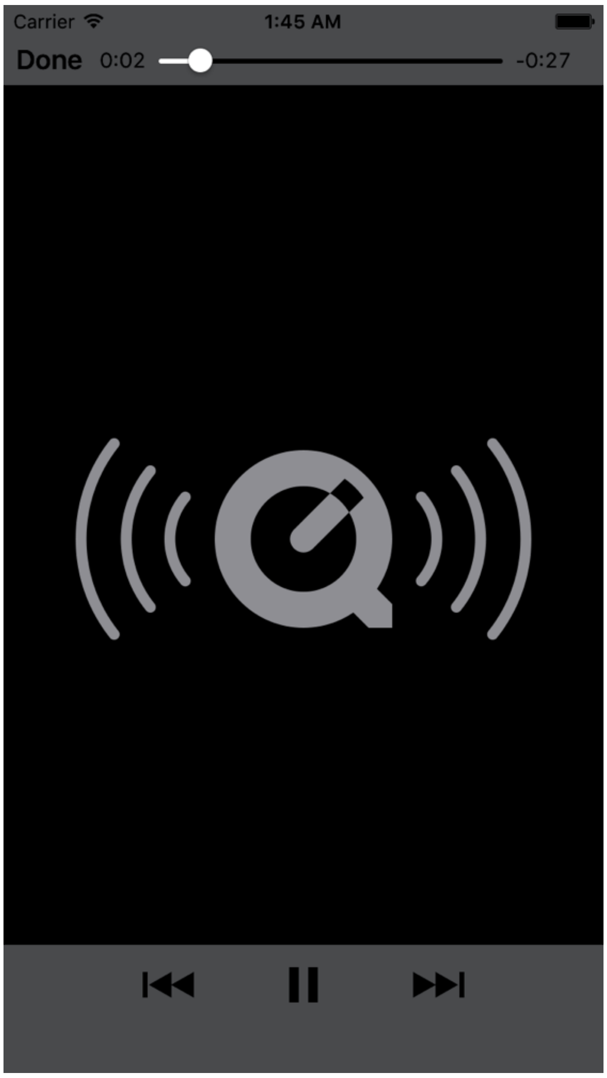
1. You extract the original request URL from the task, look up the corresponding **`Download`** in your active downloads, and remove it from that dictionary.
2. You then pass the URL to the **`localFilePath(for:)`** helper method in ***SearchViewController.swift***, which generates a permanent local file path to save to, by appending the **`lastPathComponent`** of the URL (the file name and extension of the file) to the path of the app's Documents directory.
3. Using **`FileManager`**, you move the downloaded file from its temporary file location to the desired destination file path, first clearing out any item at that location before you start the copy task. You also set the download track's **`downloaded`** property to **`true`**.
4. Finally, you use the download track's **`index`** property to reload the corresponding cell.

Build and run your project. Run a query, then pick any track and download it. When the download has finished, you'll see the file path location printed to your console:



```
file:///Users/kentoh/Library/Developer/CoreSimulator/Devices/70905975-E199-4FCF-
B238-0582AC60A7A7/data/Containers/Data/Application/FBF107E4-B8A3-4879-8790-E89570C95D4A/
Documents/mzaf_7991652075174454658.plus.aac.p.m4a
```

The Download button disappears now, because the delegate method set the track's **`downloaded`** property to **`true`**. Tap the track and you'll hear it play in the presented **`AVPlayerViewController`** as shown below:



Pausing, Resuming and Cancelling Downloads

What if the user wants to pause a download, or cancel it altogether? In this section, you'll implement the pause, resume and cancel features to give the user complete control over the download process.

You'll start by allowing the user to cancel an active download.

In ***DownloadService.swift***, replace the **`cancelDownload(_:)`** stub with the following code:

```
func cancelDownload(_ track: Track) {
  if let download = activeDownloads[track.previewURL] {
    download.task?.cancel()
    activeDownloads[track.previewURL] = nil
  }
}
```

To cancel a download, you retrieve the download task from the corresponding **`Download`** in the dictionary of active downloads, and call **`cancel()`** on it to cancel the task. You then remove the download object from the dictionary of active downloads.

Pausing a download is conceptually similar to cancelling: pausing cancels the download task, but also produces ***resume data***, which contains enough information to resume the download at a later time, if the host server supports that functionality.

Note: You can only resume a download under certain conditions. For instance, the resource must not have changed since you first requested it. For a full list of conditions, check out the documentation here.

Now, replace the **`pauseDownload(_:)`** stub with the following code:

```
func pauseDownload(_ track: Track) {
  guard let download = activeDownloads[track.previewURL] else { return
  if download.isDownloading {
    download.task?.cancel(byProducingResumeData: { data in
      download.resumeData = data
    })
    download.isDownloading = false
  }
}
```

The key difference here is you call **`cancel(byProducingResumeData:)`** instead of **`cancel()`**. You provide a closure parameter to this method, where you save the resume data to the appropriate **`Download`** for future resumption.

You also set the **`isDownloading`** property of the **`Download`** to **`false`** to indicate that the download is paused.

With the pause function completed, the next order of business is to allow the resumption of a paused download.

Replace the **`resumeDownload(_:)`** stub with the following code:

```
func resumeDownload(_ track: Track) {
  guard let download = activeDownloads[track.previewURL] else { return
  if let resumeData = download.resumeData {
    download.task = downloadsSession.downloadTask(withResumeData: resu
  } else {
    download.task = downloadsSession.downloadTask(with: download.track
  }
  download.task!.resume()
  download.isDownloading = true
}
```

When the user resumes a download, you check the appropriate **`Download`** for the presence of resume data. If found, you create a new download task by invoking **`downloadTask(withResumeData:)`** with the resume data. If the resume data is absent for some reason, you create a new download task with the download URL.

In both cases, you start the task by calling **`resume()`**, and set the **`isDownloading`** flag of the **`Download`** to **`true`**, to indicate the download has resumed.

There’s only one thing left to do for these three functions to work properly: you need to show or hide the Pause/Resume and Cancel buttons, as appropriate. To do this, the **`TrackCell`** **`configure(track:downloaded:)`** method needs to know if the track has an active download, and whether it’s currently downloading.

In ***TrackCell.swift***, change **`configure(track:downloaded:)`** to **`configure(track:downloaded:download:)`**:

```
func configure(track: Track, downloaded: Bool, download: Download?) {
```

In ***SearchViewController.swift***, fix the call in **`tableView(_:cellForRowAt:)`**:

```
cell.configure(track: track, downloaded: track.downloaded,
  download: downloadService.activeDownloads[track.previewURL])
```

Here, you extract the track’s download object from the **`activeDownloads`** dictionary.

Back in ***TrackCell.swift***, locate the two TODOs in **`configure(track:downloaded:download:)`**. Replace the first **`// TODO`** with this property:

```
var showDownloadControls = false
```

And replace the second **`// TODO`** with the following code:

```
if let download = download {
  showDownloadControls = true
  let title = download.isDownloading ? "Pause" : "Resume"
  pauseButton.setTitle(title, for: .normal)
}
```

As the comment notes, a non-nil download object means a download is in progress, so the cell should show the download controls: Pause/Resume and Cancel. Since the pause and resume functions share the same button, you toggle the button between the two states, as appropriate.

Below this if-closure, add the following code:

```
pauseButton.isHidden = !showDownloadControls
cancelButton.isHidden = !showDownloadControls
```

Here, you show the buttons for a cell only if a download is active.

Finally, replace the last line of this method:

```
downloadButton.isHidden = downloaded
```

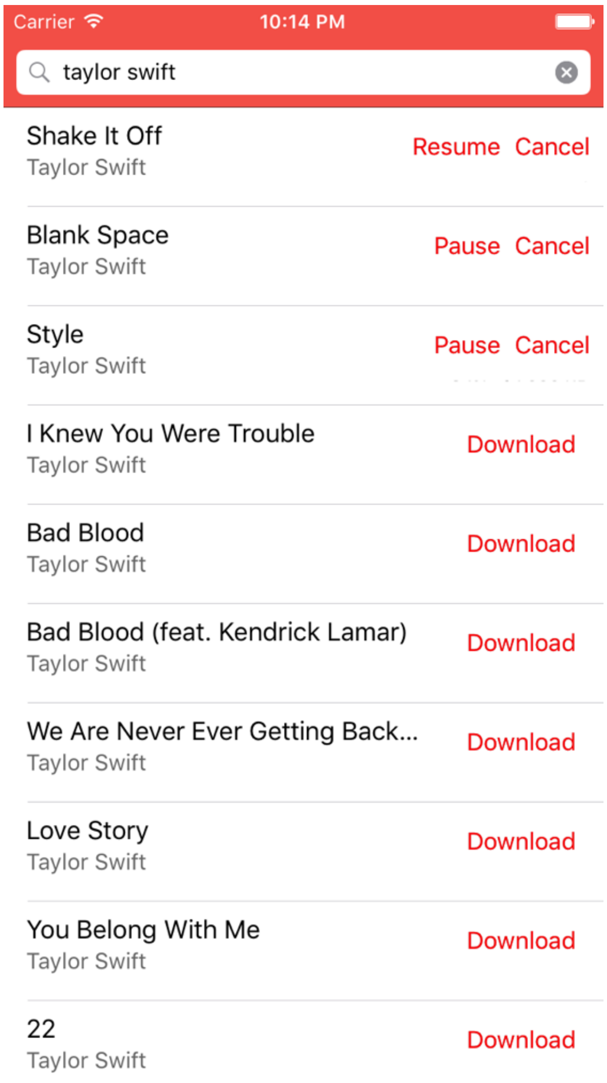
with the following code:

```
downloadButton.isHidden = downloaded || showDownloadControls
```

Here, you tell the cell to hide the Download button if its track is downloading.

Build and run your project; download a few tracks concurrently and you'll be able to pause, resume and cancel them at will:

Note: If your downloads hang after you tap Resume, tap Pause, then Resume again. This is a mysterious bug that disappears when you change the download session configuration to **`URLSessionConfiguration.background(withIdentifier: "bgSessionConfiguration")`**



Showing Download Progress

Currently, the app doesn't show the progress of the download. To improve the user experience, you'll change your app to listen for download progress events, and display the progress in the cells. And there's a session delegate method that's perfect for this job!

First, in ***TrackCell.swift***, add the following helper method:

```
func updateDisplay(progress: Float, totalSize : String) {
  progressView.progress = progress
  progressLabel.text = String(format: "%.1f%% of %@", progress * 100,
}
```

The track cell has **`progressView`** and **`progressLabel`** outlets. The delegate method will call this helper method to set their values.

Next, in ***SearchVC+URLSessionDelegates.swift***, add the following delegate method to the **`URLSessionDownloadDelegate`** extension:

```
func urlSession(_ session: URLSession, downloadTask: URLSessionDownloa
  didWriteData bytesWritten: Int64, totalBytesWritten: Int64,
  totalBytesExpectedToWrite: Int64) {
  // 1
  guard let url = downloadTask.originalRequest?.url,
    let download = downloadService.activeDownloads[url]  else { return
  // 2
  download.progress = Float(totalBytesWritten) / Float(totalBytesExpec
  // 3
  let totalSize = ByteCountFormatter.string(fromByteCount: totalBytesE
  // 4
    DispatchQueue.main.async {
    if let trackCell = self.tableView.cellForRow(at: IndexPath(row: dc
      section: 0)) as? TrackCell {
      trackCell.updateDisplay(progress: download.progress, totalSize:
    }
  }
}
```

Looking through this delegate method, step-by-step:

1. You extract the URL of the provided **`downloadTask`**, and use it to find the matching **`Download`** in your dictionary of active downloads.
2. The method also provides the total bytes written and the total bytes expected to be written. You calculate the progress as the ratio of these two values, and save the result in the **`Download`**.

The track cell will use this value to update the progress view.

3. **`ByteCountFormatter`** takes a byte value and generates a human-readable string showing the total download file size. You'll use this string to show the size of the download alongside the percentage complete.
4. Finally, you find the cell responsible for displaying the **`Track`**, and call the cell's helper method to update its progress view and progress label with the values derived from the previous steps. This involves the UI, so you do it on the main queue.

Now, update the cell's configuration, to properly display the progress view and status when a download is in progress.

Open ***TrackCell.swift***. In **`configure(track:downloaded:download:)`**, add the following line *inside* the if-closure, after the pause button title is set:

```
progressLabel.text = download.isDownloading ? "Downloading..." : "Paus
```

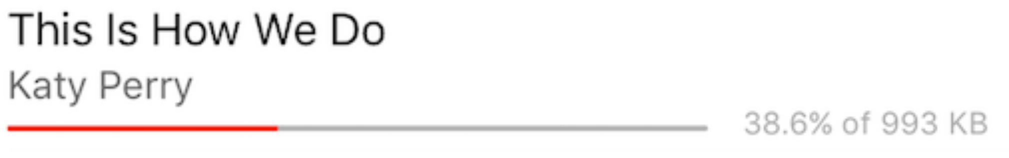
This gives the cell something to show, before the first update from the delegate method, and while the download is paused.

And add the following code *below* the if-closure, below the **`isHidden`** lines for the two buttons:

```
progressView.isHidden = !showDownloadControls
progressLabel.isHidden = !showDownloadControls
```

As for the buttons, this shows the progress view and label only while the download is in progress.

Build and run your project; download any track and you should see the progress bar status update as the download progresses:



Hurray, you've made, erm, progress! :]

Enabling Background Transfers

Your app is quite functional at this point, but there's one major enhancement left to add: background transfers. In this mode, downloads continue even when your app is backgrounded or crashes for any reason. This isn't really necessary for song snippets, which are pretty small; but your users will appreciate this feature if your app transfers large files.

But if your app isn't running, how can this work? The OS runs a separate daemon outside the app to manage background transfer tasks, and it sends the appropriate delegate messages to the app as the download tasks run. In the event the app terminates during an active transfer, the tasks will continue to run unaffected in the background.

When a task completes, the daemon will relaunch the app in the background. The re-launched app will re-create the background session, to receive the relevant completion delegate messages, and perform any required actions such as persisting downloaded files to disk.

Note: If the user terminates the app by force-quiting from the app switcher, the system will cancel all of the session's background transfers, and won't attempt to relaunch the app.

You access this magic by creating a session with the ***background*** session configuration.

In ***SearchViewController.swift***, in the initialization of **`downloadsSession`**, find the following line of code:

```
let configuration = URLSessionConfiguration.default
```

…and replace it with the following line:

```
let configuration = URLSessionConfiguration.background(withIdentifier:
  "bgSessionConfiguration")
```

Instead of using a default session configuration, you use a special background session configuration. Note that you also set a unique identifier for the session here to allow your app to create a new background session, if needed.

Note: You must not create more than one session for a background configuration, because the system uses the *configuration*'s identifier to associate tasks with the *session*.

If a background task completes when the app isn't running, the app will be relaunched in the background. You'll need to handle this event from your app delegate.

Switch to ***AppDelegate.swift***, and add the following code near the top of the class:

```
var backgroundSessionCompletionHandler: (() -> Void)?
```

Next, add the following method to ***AppDelegate.swift***:

```
func application(_ application: UIApplication, handleEventsForBackgrou
  identifier: String, completionHandler: @escaping () -> Void) {
  backgroundSessionCompletionHandler = completionHandler
}
```

Here, you save the provided **`completionHandler`** as a variable in your app delegate for later use.

`application(_:handleEventsForBackgroundURLSession:)` wakes up the app to deal with the completed background task. You need to handle two things in this method:

- First, the app needs to re-create the appropriate background configuration and session, using the identifier provided by this delegate method. But since this app creates the background session when it instantiates **`SearchViewController`**, you're already reconnected at this point!
- Second, you'll need to capture the completion handler provided by this delegate method. Invoking the completion handler tells the OS that your app's done working with all background activities for the current session, and also causes the OS to snapshot your updated UI for display in the app switcher.

The place to invoke the provided completion handler is **`urlSessionDidFinishEvents(forBackgroundURLSession:)`**: it's a **`URLSessionDelegate`** method that fires when all tasks pertaining to the background session have finished.

In ***SearchVC+URLSessionDelegates.swift*** find the import:

```
import Foundation
```

and add the following import underneath:

```
import UIKit
```

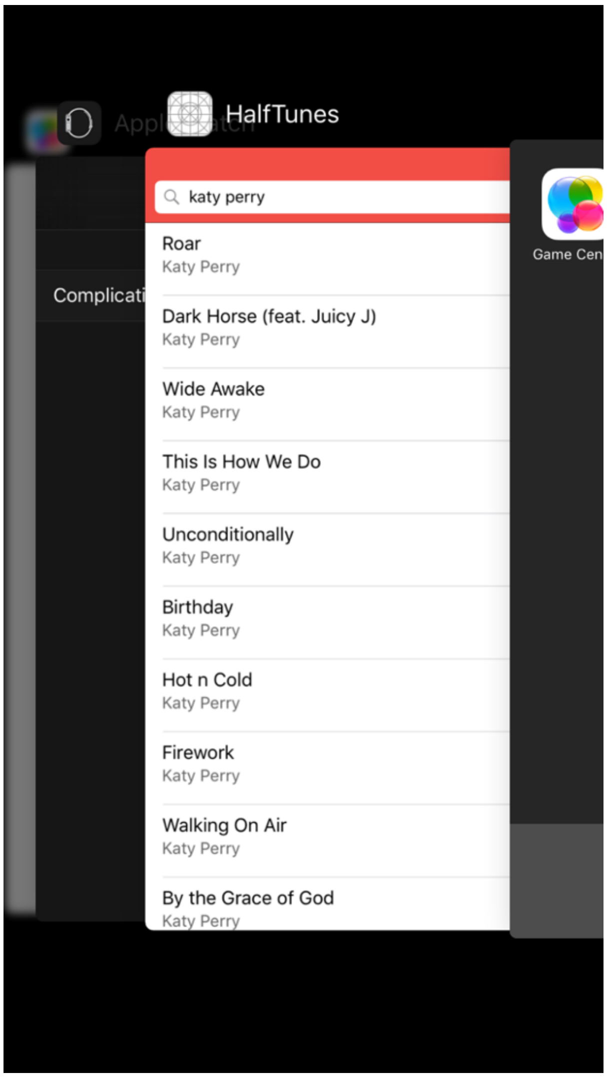
lastly, add the following extension:

```
extension SearchViewController: URLSessionDelegate {

  // Standard background session handler
  func urlSessionDidFinishEvents(forBackgroundURLSession session: URLS
    DispatchQueue.main.async {
      if let appDelegate = UIApplication.shared.delegate as? AppDelega
        let completionHandler = appDelegate.backgroundSessionCompletic
        appDelegate.backgroundSessionCompletionHandler = nil
        completionHandler()
      }
    }
  }

}
```

The above code simply grabs the stored completion handler from the app delegate and invokes it on the main thread. You reference the app delegate by getting the shared delegate from the UIApplication, which is accessible thanks to the UIKit import.

Build and run your app; start a few concurrent downloads and tap the ***Home*** button to background the app. Wait until you think the downloads have completed, then double-tap the Home button to reveal the app switcher.

The downloads should have finished, with their new status reflected in the app snapshot. Open the app to confirm this:



katy perry

Roar
Katy Perry

Dark Horse (feat. Juicy J)
Katy Perry

Wide Awake
Katy Perry

This Is How We Do
Katy Perry

Unconditionally
Katy Perry

Birthday
Katy Perry

Hot n Cold
Katy Perry

Firework
Katy Perry

Walking On Air
Katy Perry

By the Grace of God
Katy Perry

You now have a fully functional music streaming app! Your move now, Apple Music! :]

Where To Go From Here?

You can download the complete project for this tutorial here.

Congratulations! You're now well-equipped to handle most common networking requirements in your app. There are more **`URLSession`** topics than would fit in this tutorial, for example, upload tasks and session configuration settings, such as timeout values and caching policies.

To learn more about these features (and others!), check out the following resources:

- Apple's URLSession Programming Guide contains comprehensive details on everything you'd want to do.
- Our own Networking with URLSession video course starts with HTTP basics, and covers tasks, background sessions, authentication, App Transport Security, architecture and unit testing.
- AlamoFire is a popular third-party iOS networking library; we covered the basics of it in our Beginning Alamofire tutorial.

I hope you found this tutorial useful. Feel free to join the discussion below!

Team

Each tutorial at www.raywenderlich.com is created by a team of dedicated developers so that it meets our high quality standards. The team members who worked on this tutorial are: