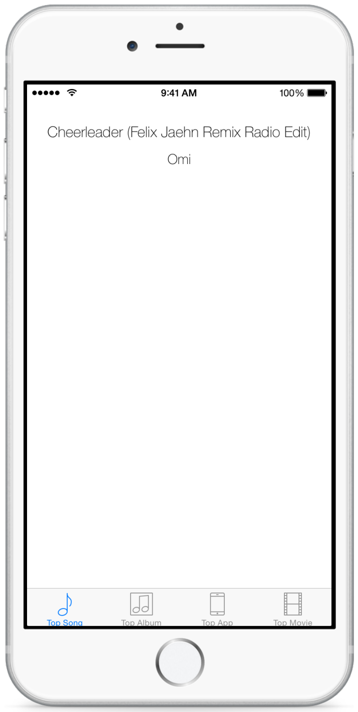
RSSReader
Lesson 6

Description

Implement the retrieval and parsing of an RSS feed to display song information.



Learning Outcomes

- Explain what an RSS feed is, and generalize the structure of XML and JSON data formats.
- Discover how an iOS app may obtain RSS data with an http request.
- Describe the use of `NSURLRequest`, `NSURLConnection`, `NSJSONSerialization`, and `NSDictionary` classes to inspect RSS feed data.
- Describe the behavior of asynchronous method calls, and relate asynchronous method calls to retrieving data over a network.
- Observe how closures may be passed to methods and invoked by the receiving method.

Vocabulary

RSS feed	HTTP	URL
`NSURL`	request	`NSURLRequest`
`NSURLConnection`	asynchronous method	closure
XML	JSON	dictionary
`NSDictionary`	type casting	optional binding

Materials

- **RSSReader Lesson 6** Xcode project
- Internet connectivity to the **ax.itunes.apple.com** domain
- RSS sample data text file (**sampledata.rss.txt**)
- **Asynchronous Methods** presentation
- **Closures** presentation

Opening

How can we obtain data from RSS feeds and display the data in our views?

Agenda

- Using a web browser, explore the Apple RSS feeds page: http://www.apple.com/rss/.
- Click on the Top 10 Songs link, and observe the xml output.
- Using the web browser, modify the url in the address bar, replacing `limit=10` with `limit=1` and replacing `/xml` with `/json`, and observe the output displayed within the browser.
- Present the RSS sample data text file if necessary.
- Explain how XML and JSON are simple structured data formats.
- Discuss how the RSS data can also be obtained from an iOS app by making a similar http request, and traversing the data structure.
- Update the `TopMediaController` `viewDidLoad` implementation with an extraction of the JSON data.

```
override func viewDidLoad() {
    super.viewDidLoad()
    let feedURL = "http://ax.itunes.apple.com/WebObjects/MZStoreServices.woa/ws/
RSS/topsongs/limit=1/json"
    let request = NSURLRequest(URL: NSURL(string: feedURL)!)
    NSURLConnection.sendAsynchronousRequest(request,
        queue: NSOperationQueue.mainQueue()) { response, data, error in
        if let jsonData = data,
            feed = (try? NSJSONSerialization.JSONObjectWithData(jsonData,
                options: .MutableContainers)) as? NSDictionary,
            title = feed.valueForKeyPath("feed.entry.im:name.label") as? String,
            artist = feed.valueForKeyPath("feed.entry.im:artist.label") as? String {
                self.titleLabel.text = title
                self.artistLabel.text = artist
        }
    }
}
```

- Run the app (⌘**R**), observe how the default label text appears briefly, and how the song title and artist names then appear.
- Explain how http requests for RSS data can be represented with an `NSURLRequest` object, and how the `NSURL` argument uses forced unwrapping.
- Explain how a request is sent to a server asynchronously with the `NSURLConnection sendAsynchronousRequest:queue:completionHandler:` method.
- Present the concept of asynchronous methods.
- Explain how the `queue:` parameter specifies the context of the run loop that the closure should execute within, and the best practice of using the `mainQueue` to execute blocks that update the interface.
- Discuss how the default labels in the view appear while the request for RSS data is sent asynchronously, and how the `completionHandler:` argument specifies a closure that is invoked once the data is obtained from the server.
- Present the concept of closures.
- Explain how, once the RSS data is retrieved, the closure casts the data to an `NSDictionary`, and uses multiple optional bindings to navigate the structured RSS data to obtain the specific pieces of data used by the app.
- Discuss what we might see on the screen when the song title is very long.
- Modify the implementation of `viewDidLoad` to simulate an arbitrarily long song title.

```
...
self.titleLabel.text = "A Very Long Song Title (Long Title Remix)"
...
```

- Run the app (⌘**R**), and observe that the song title does not fit within the bounds of the screen.
- Using Interface Builder, select the Title label within the Top Song scene, and use the Attributes Inspector (⌥⌘**4**) to set the Autoshrink attribute to a **Minimum Font Size** of **10**.
- Discuss how labels must have width constraints in order to infer when text content should shrink.
- Using Interface Builder, select the Title label within the Top Song scene and drag its left and right edges to the margin guides within the containing view.
- Add leading and trailing edge constraints to the Title label by Control-dragging both leftward and rightward from the label to the containing view.
- Repeat the modification of each Title label Autoshrink attribute and the addition of constraints within each scene.
- Run the app (⌘**R**), and observe that the song title text size appears smaller, to accommodate the longer song title.

- Run the app again (⌘R) and observe the default **Title** label text appearing, and discuss how the data is still being retrieved before the label text is updated by the controller.
- Explain how the labels are updated with data once the `completionHandler:` closure is invoked.
- Using Interface Builder, select each text label, use the Attributes Inspector (⌥⌘4) to ensure the Hidden attribute is checked, and observe how the labels appear lighter within the canvas.
- Update the implementation of the `TopMediaController` `viewDidLoad` method to enable the display of each label once the data is obtained.

```
...
self.titleLabel.text = title
self.titleLabel.hidden = false
self.artistLabel.text = artist
self.artistLabel.hidden = false
...
```

- Run the app (⌘R), observe the song label text appear on the Top Song tab. Interact with the other tabs, and notice how each view also displays the same top song data.

Closing

What if we were requesting data that took a very long time to retrieve, perhaps due to poor network performance? How would the user experience be affected? What should the app do if the data can not be retrieved?

Modifications and Extensions

- Use Interface Builder to add an Activity Indicator View to each interface that animates while the data is being retrieved, and then disappears once the view is updated with data.
- Carry out appropriate error handling with the `completionHandler:` closure and the `JSONObjectWithData:options:error:` methods.
- Investigate the `reduce` function, and use `reduce` to extract the data from the feed data dictionary instead of using `valueForKeyPath:`.

Resources

Apple RSS Feeds http://www.apple.com/rss/

NSURL Class Reference https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSURL_Class/

NSURLRequest Class Reference https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSURLRequest_Class/

NSURLConnection Class Reference https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSURLConnection_Class/index.html

NSOperationQueue Class Reference https://developer.apple.com/library/ios/documentation/Cocoa/Reference/NSOperationQueue_class/index.html

NSJSONSerialization Class Reference https://developer.apple.com/library/ios/documentation/Foundation/Reference/NSJSONSerialization_Class/

NSDictionary Class Reference https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSDictionary_Class/index.html

The Swift Programming Language: Type Casting https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TypeCasting.html

The Swift Programming Language: Control Flow https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html

The Swift Programming Language: Closures https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Closures.html