Web Services-3

Over the next few assignments you will be building an application called SooGreyhounds that will provide a searchable photo gallery to users.. This assignment will lay the foundation and focus on implementing the web service requests responsible for fetching the metadata for photos as well as downloading the image data for a specific photo.

1. Create a new Single View Application for the Universal device family. Name this application "SooGreyhoundsMobile" and give it an identifier of "com.soogreyhounds".

2. Create a new repository on GitHub and add your instructors as collaborators. Now go into Xcode, open the source control navigator, and add your new repository as a remote.

3. Let's knock out the basic UI before focusing on web services. Create a new Swift file named PhotosViewController. In PhotosViewController.swift, define the PhotosViewController class and give it an imageView property:

   ```
   ~~import Foundation~~
   import UIKit

   class PhotosViewController: UIViewController {

     @IBOutlet var imageView: UIImageView!

   }
   ```

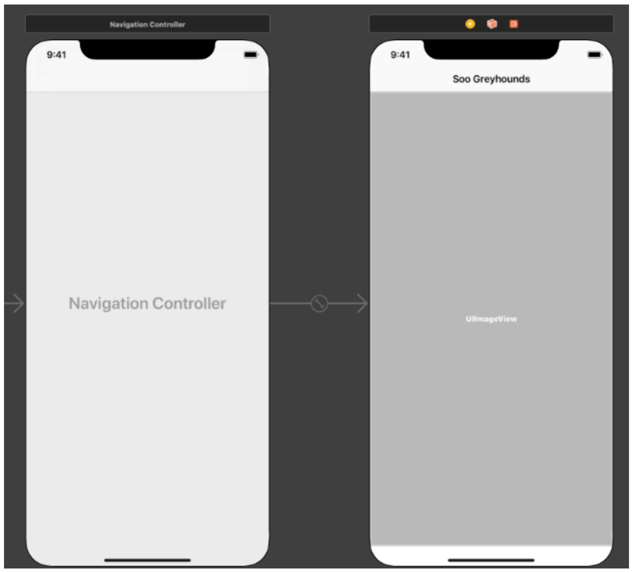
4. In the project navigator, delete the existing ViewController.swift that was created for you by Xcode (this is done as clean up, it is best to remove files you do not need).

5. Open Main.storyboard and select the View Controller. Open its identity inspector and change the Class to PhotosViewController. With the Photos View Controller still selected, select the Editor menu and choose Embed In → Navigation Controller.

6. Select the Navigation Controller and open its attributes inspector. Under the View Controller heading, make sure the box for Is Initial View Controller is checked.

7. Drag an Image View onto the canvas for PhotosViewController and add constraints to pin it to all edges of the superview. Connect the image view to the imageView outlet on PhotosViewController. Open the attributes inspector for the image view and change the Content Mode to Aspect Fill.

8. Click the navigation bar for the Photos View Controller, open the attributes inspector, and give it a title of "Soo Greyhounds." Your interface should look like this:



9. Build and run the application to make sure there are no errors.

10. Create a new Swift file named SooGreyhoundsAPI and declare the SooGreyhoundsAPI struct, which will contain all of the knowledge that is specific to the SooGreyhounds API:

```
import Foundation

struct SooGreyhoundsAPI {

}
```

11. You are going to use an enumeration to specify which endpoint on the Soo Greyhounds server to hit. For this application, you will only be working with the endpoint to get latest photos. However, Soo Greyhounds supports many additional APIs, such as searching for images based on a string. Using an enum now will make it easier to add endpoints in the future. In SooGreyhoundsAPI.swift, create the Method enumeration. Each case of Method has a raw value that matches the corresponding Soo Greyhounds endpoint:

```
import Foundation

enum Method: String {
   case latestPhotos = "soogreyhounds.photos.getList"
}

struct SooGreyhoundsAPI {

}
```

12. Now declare a type-level property to reference the base URL string for the web service requests.

```
enum Method: String {
   case latestPhotos = "soogreyhounds.photos.getList"
}

struct SooGreyhoundsAPI {
   static let baseURLString = "https://api.soogreyhounds.gutty.ca/services/rest"
}
```

13. The baseURLString is an implementation detail of the SooGreyhoundsAPI type, and no other type needs to know about it. Instead, they will ask for a completed URL from SooGreyhoundsAPI. To keep other files from being able to access baseURLString, mark the property as private.

```
struct SooGreyhoundsAPI {
   private static let baseURLString = "https://api.soogreyhounds.gutty.ca/services/rest"
}
```

14. Now you are going to create a type method that builds up the Soo Greyhounds URL for a specific endpoint. This method will accept two arguments: The first will specify which endpoint to hit using the Method enumeration, and the second will be an optional dictionary of query item parameters associated with the request. Implement this method in your SooGreyhoundsAPI struct in SooGreyhoundsAPI.swift. For now, this method will return an empty URL:

```
private static func sooGreyhoundsURL(method: Method, parameters: [String:String]?) -> URL {
   return URL(string: "")!
}
```

15. In SooGreyhoundsAPI.swift, define and implement the latestPhotosURL computed property.

```
static var latestPhotosURL: URL {
   return sooGreyhoundsURL(method: .latestPhotos,
                parameters: ["extras": "url_h,date_taken"])
}
```

16. Notice that the sooGreyhoundsURL(method:parameters:) method is private. It is an implementation detail of the SooGreyhoundsAPI struct. An internal type method will be exposed to the rest of the project for each of the specific endpoint URLs (currently, just the latest photos endpoint). These internal type methods will call through to the sooGreyhoundsURL(method:parameters:) method. In SooGreyhoundsAPI.swift, define and implement the latestPhotosURL computed property:

```
private static func sooGreyhoundsURL(method: Method, parameters: [String:String]?) -> URL {
~~  return URL(string: "")!~~

  var components = URLComponents(string: baseURLString)!

  var queryItems = [URLQueryItem]()

  if let additionalParams = parameters {
    for (key, value) in additionalParams {
      let item = URLQueryItem(name: key, value: value)
      queryItems.append(item)
    }
  }
  components.queryItems = queryItems

  return components.url!
}
```

17. The last step in setting up the URL is to pass in the parameters that are common to all requests: method, api_key, format, and nojsoncallback. The API key is a token generated by Soo Greyhounds to identify your application and authenticate it with the web service. We have generated an API key for this application by creating a Soo Greyhounds account and registering this application. In SooGreyhoundsAPI.swift, create a constant that references this token.

```
struct SooGreyhoundsAPI {
  private static let baseURLString = "https://api.soogreyhounds.gutty.ca/services/rest"
  private static let apiKey = "a6d819499131071f158fd740860a5a88"
```

18. Finish implementing sooGreyhoundsURL(method:parameters:) to add the common query items to the URLComponents.

```
private static func sooGreyhoundsURL(method: Method, parameters: [String:String]?) -> URL {
    var components = URLComponents(string: baseURLString)!
    var queryItems = [URLQueryItem]()

    let baseParams = [
        "method": method.rawValue,
        "format": "json",
        "nojsoncallback": "1",
        "api_key": apiKey
    ]

    for (key, value) in baseParams {
        let item = URLQueryItem(name: key, value: value)
        queryItems.append(item)
    }

    …

    return components.url!
}
```

19. In SooGreyhounds, a new class, PhotoStore, will be responsible for initiating the web service requests. It will use the URLSession API and the SooGreyhoundsAPI struct to fetch a list of latest photos and download the image data for each photo. Create a new Swift file named PhotoStore and declare the PhotoStore class:

```
import Foundation
class PhotoStore {

}
```

20. In PhotoStore.swift, add a property to hold on to an instance of URLSession.

```
class PhotoStore {
    private let session: URLSession = {
        let config = URLSessionConfiguration.default
        return URLSession(configuration: config)
    }()
}
```

21. In PhotoStore.swift, implement the fetchLatestPhotos() method to create a URLRequest that connects to api.soogreyhounds.gutty.ca and asks for the list of latest photos. Then, use the URLSession to create a URLSessionDataTask that transfers this request to the server:

```
func fetchLatestPhotos() {
   let url = SooGreyhoundsAPI.latestPhotosURL
   let request = URLRequest(url: url)
   let task = session.dataTask(with: request) {
      (data, response, error) -> Void in

      if let jsonData = data {
         if let jsonString = String(data: jsonData,
                                encoding: .utf8) {
            print(jsonString)
         }
      } else if let requestError = error {
         print("Error fetching latest photos: \(requestError)")
      } else {
         print("Unexpected error with the request")
      }
   }
   task.resume()
}
```

22. To make a request, PhotosViewController will call the appropriate methods on PhotoStore. To do this, PhotosViewController needs a reference to an instance of PhotoStore. At the top of PhotosViewController.swift, add a property to hang on to an instance of PhotoStore:

```
class PhotosViewController: UIViewController {
   @IBOutlet var imageView: UIImageView!
   var store: PhotoStore!
```

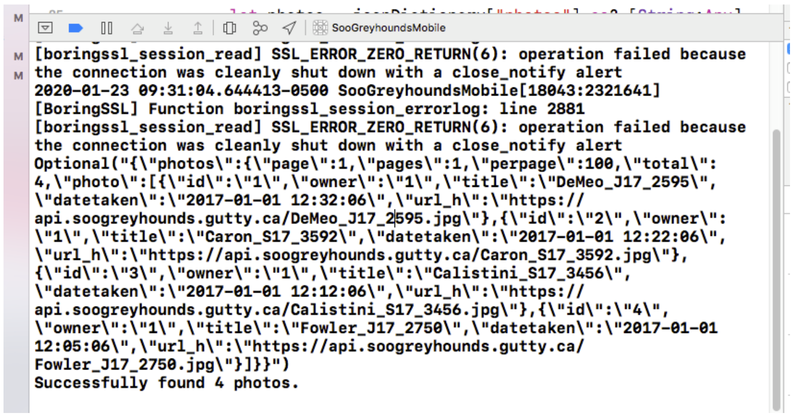
23. The store is a dependency of the PhotosViewController. You will use property injection to give the PhotosViewController its store dependency, just as you did with the view controllers in Home Owner. Open AppDelegate.swift and use property injection to give the PhotosViewController an instance of PhotoStore:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
      launchOptions: [UIApplicationLaunchOptionsKey : Any]?) -> Bool {
   let rootViewController = window!.rootViewController as! UINavigationController
   let photosViewController =
      rootViewController.topViewController as! PhotosViewController
   photosViewController.store = PhotoStore()
   return true
}
```

24. Now that the PhotosViewController can interact with the PhotoStore, kick off the web service exchange when the view controller is coming onscreen for the first time. In PhotosViewController.swift, override viewDidLoad() and fetch the latest photos:

```
override func viewDidLoad() {
   super.viewDidLoad()
   store.fetchLatestPhotos()
}
```

25. Build and run the application. A string representation of the JSON data coming back from the web service will print to the console. (If you do not see anything print to the console, make sure you typed the URL and API key correctly.) The response will look something like this:



```
[boringssl_session_read] SSL_ERROR_ZERO_RETURN(6): operation failed because
the connection was cleanly shut down with a close_notify alert
2020-01-23 09:31:04.644413-0500 SooGreyhoundsMobile[18043:2321641]
[BoringSSL] Function boringssl_session_errorlog: line 2881
[boringssl_session_read] SSL_ERROR_ZERO_RETURN(6): operation failed because
the connection was cleanly shut down with a close_notify alert
Optional("{\"photos\":{\"page\":1,\"pages\":1,\"perpage\":100,\"total\":
4,\"photo\":[{\"id\":\"1\",\"owner\":\"1\",\"title\":\"DeMeo_J17_2595\",
\"datetaken\":\"2017-01-01 12:32:06\",\"url_h\":\"https://
api.soogreyhounds.gutty.ca/DeMeo_J17_2595.jpg\"},{\"id\":\"2\",\"owner\":
\"1\",\"title\":\"Caron_S17_3592\",\"datetaken\":\"2017-01-01 12:22:06\",
\"url_h\":\"https://api.soogreyhounds.gutty.ca/Caron_S17_3592.jpg\"},
{\"id\":\"3\",\"owner\":\"1\",\"title\":\"Calistini_S17_3456\",
\"datetaken\":\"2017-01-01 12:12:06\",\"url_h\":\"https://
api.soogreyhounds.gutty.ca/Calistini_S17_3456.jpg\"},{\"id\":\"4\",
\"owner\":\"1\",\"title\":\"Fowler_J17_2750\",\"datetaken\":\"2017-01-01
12:05:06\",\"url_h\":\"https://api.soogreyhounds.gutty.ca/
Fowler_J17_2750.jpg\"}]}}")
Successfully found 4 photos.
```

26. Next, you will create a Photo class to represent each photo that is returned from the web service request. The relevant pieces of information that you will need for this application are the id, the title, the url_h, and the datetaken. Create a new Swift file called Photo and declare the Photo class with properties for the photoID, the title, and the remoteURL. Finally, add a designated initializer that sets up the instance:

```
import Foundation
class Photo {
   let title: String
   let remoteURL: URL
   let photoID: String
   let dateTaken: Date

   init(title: String, photoID: String, remoteURL: URL, dateTaken: Date) {
      self.title = title
      self.photoID = photoID
      self.remoteURL = remoteURL
      self.dateTaken = dateTaken
   }
}
```

27. Apple has a built-in class for parsing JSON data, JSONSerialization. You can hand this class a bunch of JSON data, and it will create a dictionary for every JSON dictionary (the JSON specification calls these “objects”), an array for every JSON array, a String for every JSON string, and an NSNumber for every JSON number. Let’s see how this class helps you. Open PhotoStore.swift and update fetchLatestPhotos() to print the JSON object to the console:

```
func fetchLatestPhotos() {

  let url = SooGreyhoundsAPI.latestPhotosURL
  let request = URLRequest(url: url)
  let task = session.dataTask(with: request) {
    (data, response, error) -> Void in

    if let jsonData = data {
      ~~if let jsonString = String(data: jsonData,~~
                     ~~encoding: .utf8) {~~
        ~~print(jsonString)~~
      ~~}~~
      do {
        let jsonObject = try JSONSerialization.jsonObject(with: jsonData,
                                   options: [])
        print(jsonObject)
      } catch let error {
        print("Error creating JSON object: \(error)")
      }
    } else if let requestError = error {
      print("Error fetching latest photos: \(requestError)")
    } else {
      print("Unexpected error with the request")
    }
  }
  task.resume()
}
```

28. Build and run the application, then check the console. You will see the JSON data again, but now it will be formatted differently because print() does a good job formatting dictionaries and arrays. The format of the JSON data is dictated by the API, so you will add the code to parse the JSON to the SooGreyhoundsAPI struct. Parsing the data that comes back from the server could go wrong in a number of ways: The data might not contain JSON. The data could be corrupt. The data might contain JSON but not match the format that you expect. To manage the possibility of failure, you will use an enumeration with associated values to represent the success or failure of the parsing.

29. In PhotoStore.swift, add an enumeration named PhotosResult to the top of the file that has a case for both success and failure:

```
import Foundation

enum PhotosResult {
  case success([Photo])
  case failure(Error)
}

class PhotoStore {
```

30. If the data is valid JSON and contains an array of photos, those photos will be associated with the success case. If there are any errors during the parsing process, the relevant Error will be passed along with the failure case. Error is a protocol that all errors conform to. NSError is the error that many iOS frameworks throw, and it conforms to Error. You will create your own Error shortly. In SooGreyhoundsAPI.swift, implement a method that takes in an instance of Data and uses the JSONSerialization class to convert the data into the basic foundation objects (This code will generate some warnings. You will resolve them shortly):

```
static func photos(fromJSON data: Data) -> PhotosResult {
  do {
    let jsonObject = try JSONSerialization.jsonObject(with: data,
                                                      options: [])

    var finalPhotos = [Photo]()
    return .success(finalPhotos)
  } catch let error {
    return .failure(error)
  }
}
```

31. At the top of SooGreyhoundsAPI.swift, declare a custom enum to represent possible errors for the Soo Greyhounds API:

```
enum SooGreyhoundsError: Error {
  case invalidJSONData
}

enum Method: String {
  case latestPhotos = "soogreyhounds.photos.getList"
}
```

32. Now, in photos(fromJSON:), dig down through the JSON data to get to the array of dictionaries representing the individual photos.

```
static func photos(fromJSON data: Data) -> PhotosResult {
    do {
        let jsonObject = try JSONSerialization.jsonObject(with: data, options: [])
        guard
            let jsonDictionary = jsonObject as? [AnyHashable:Any],
            let photos = jsonDictionary["photos"] as? [String:Any],
            let photosArray = photos["photo"] as? [[String:Any]] else {

            // The JSON structure doesn't match our expectations
            return .failure(SooGreyhoundsError.invalidJSONData)
        }
    …
}
```

33. The next step is to get the photo information out of the dictionary and into Photo model objects. You will need an instance of DateFormatter to convert the datetaken string into an instance of Date. In SooGreyhoundsAPI.swift, add a constant instance of DateFormatter:

```
private static let apiKey = "a6d819499131071f158fd740860a5a88"
private static let dateFormatter: DateFormatter = {
    let formatter = DateFormatter()
    formatter.dateFormat = "yyyy-MM-dd HH:mm:ss"
    return formatter
}()
```

34. Still in SooGreyhoundsAPI.swift, write a new method to parse a JSON dictionary into a Photo instance:

```
private static func photo(fromJSON json: [String : Any]) -> Photo? {
    guard
        let photoID = json["id"] as? String,
        let title = json["title"] as? String,
        let dateString = json["datetaken"] as? String,
        let photoURLString = json["url_h"] as? String,
        let url = URL(string: photoURLString),
        let dateTaken = dateFormatter.date(from: dateString) else {

            // Don't have enough information to construct a Photo
            return nil
    }
    return Photo(title: title, photoID: photoID, remoteURL: url, dateTaken: dateTaken)
}
```

35. Now update photos(fromJSON:) to parse the dictionaries into Photo instances and then return these as part of the success enumerator. Also handle the possibility that the JSON format has changed, so no photos were able to be found:

```
static func photos(fromJSON data: Data) -> PhotosResult {
    ….
    var finalPhotos = [Photo]()
    for photoJSON in photosArray {
        if let photo = photo(fromJSON: photoJSON) {
            finalPhotos.append(photo)
        }
    }

    if finalPhotos.isEmpty && !photosArray.isEmpty {
        // We weren't able to parse any of the photos
        // Maybe the JSON format for photos has changed
        return .failure(SooGreyhoundsError.invalidJSONData)
    }
    …
}
```

36. Next, in PhotoStore.swift, write a new method that will process the JSON data that is returned from the web service request.

```
private func processPhotosRequest(data: Data?, error: Error?) -> PhotosResult {
    guard let jsonData = data else {
        return .failure(error!)
    }

    return SooGreyhoundsAPI.photos(fromJSON: jsonData)
}
```

37. Now, update fetchLatestPhotos() to use the method you just created:

```
func fetchLatestPhotos() {
   …
   let task = session.dataTask(with: request) {
      (data, response, error) -> Void in

      ~~if let jsonData = data {~~
         ~~do {~~
            ~~let jsonObject = try JSONSerialization.jsonObject(with: jsonData,~~
                                                 ~~options: [])~~
            ~~print(jsonObject)~~
         ~~} catch let error {~~
            ~~print("Error creating JSON object: \(error)")~~
         ~~}~~
      ~~} else if let requestError = error {~~
         ~~print("Error fetching latest photos: \(requestError)")~~
      ~~} else {~~
         ~~print("Unexpected error with the request")~~
      ~~}~~

      **let result = self.processPhotosRequest(data: data, error: error)**
   }
   task.resume()
}
```

38. Update the method signature for fetchLatestPhotos() to take in a completion closure that will be called once the web service request is completed:

```
func fetchLatestPhotos(**completion: @escaping (PhotosResult) -> Void**) {
   let url = SooGreyhoundsAPI.latestPhotosURL
   let request = URLRequest(url: url)
   let task = session.dataTask(with: request) {
      (data, response, error) -> Void in

      let result = self.processPhotosRequest(data: data, error: error)
      **completion(result)**
   }
   task.resume()
```

39. In PhotosViewController.swift, update the implementation of the viewDidLoad() using the trailing closure syntax to print out the result of the web service request:

```
override func viewDidLoad() {
   super.viewDidLoad()

   store.fetchLatestPhotos ~~()~~ {
      (photosResult) -> Void in

      switch photosResult {
      case let .success(photos):
         print("Successfully found \(photos.count) photos.")
      case let .failure(error):
         print("Error fetching latest photos: \(error)")
      }
   }
}
```

40. Build and run the application. Once the web service request completes, you should see the number of photos found printed to the console.

41. Now, you will use the URL returned from the web service request to download the image data. Then you will create an instance of UIImage from that data, and, finally, you will display the first image returned from the request in a UIImageView. (In the next chapter, you will display all of the images that are returned in a grid layout driven by a UICollectionView.) The first step is downloading the image data. This process will be very similar to the web service request to download the photos' JSON data. Open PhotoStore.swift, import UIKit, and add an enumeration to the top of the file that represents the result of downloading the image. This enumeration will follow the same pattern as the PhotosResult enumeration, taking advantage of associated values. You will also create an Error to represent photo errors:

```
~~import Foundation~~
import UIKit

enum ImageResult {
   case success(UIImage)
   case failure(Error)
}
enum PhotoError: Error {
   case imageCreationError
}
enum PhotosResult {
   case success([Photo])
   case failure(Error)
}
```

42. If the download is successful, the success case will have the UIImage associated with it. If there is an error, the failure case will have the Error associated with it. Now, in the same file, implement a method to download the image data. Like the fetchLatestPhotos(completion:) method, this new method will take in a completion closure that will return an instance of ImageResult:

```
func fetchImage(for photo: Photo, completion: @escaping (ImageResult) -> Void) {

  let photoURL = photo.remoteURL
  let request = URLRequest(url: photoURL)

  let task = session.dataTask(with: request) {
    (data, response, error) -> Void in

  }
  task.resume()
}
```

43. Now implement a method that processes the data from the web service request into an image, if possible:

```
private func processImageRequest(data: Data?, error: Error?) -> ImageResult {
  guard
    let imageData = data,
    let image = UIImage(data: imageData) else {

      // Couldn't create an image
      if data == nil {
        return .failure(error!)
      } else {
        return .failure(PhotoError.imageCreationError)
      }
  }

  return .success(image)
}
```

44. Still in PhotoStore.swift, update fetchImage(for:completion:) to use this new method:

```
func fetchImage(for photo: Photo, completion: @escaping (ImageResult) -> Void) {

   let photoURL = photo.remoteURL
   let request = URLRequest(url: photoURL)

   let task = session.dataTask(with: request) {
      (data, response, error) -> Void in

      let result = self.processImageRequest(data: data, error: error)
      completion(result)
   }
   task.resume()
}
```

45. To test this code, you will download the image data for the first photo that is returned from the latest photos request and display it on the image view. Open PhotosViewController.swift and add a new method that will fetch the image and display it on the image view:

```
func updateImageView(for photo: Photo) {
   store.fetchImage(for: photo) {
      (imageResult) -> Void in

      switch imageResult {
      case let .success(image):
         self.imageView.image = image
      case let .failure(error):
         print("Error downloading image: \(error)")
      }
   }
}
```

46. Now update viewDidLoad() to use this new method:

```
override func viewDidLoad() {
    super.viewDidLoad()

    store.fetchLatestPhotos {
        (photosResult) -> Void in

        switch photosResult {
        case let .success(photos):
            print("Successfully found \(photos.count) photos.")
            if let firstPhoto = photos.first {
                self.updateImageView(for: firstPhoto)
            }
        case let .failure(error):
            print("Error fetching latest photos: \(error)")
        }
    }
}
```

47. Next, you will update the asynchronous PhotoStore methods to call their completion handlers on the main thread. In PhotoStore.swift, update fetchLatestPhotos(completion:) to call the completion closure on the main thread:

```
func fetchLatestPhotos(completion: @escaping (PhotosResult) -> Void) {

    let url = SooGreyhoundsAPI.latestPhotosURL
    let request = URLRequest(url: url)
    let task = session.dataTask(with: request) {
        (data, response, error) -> Void in

        let result = self.processPhotosRequest(data: data, error: error)
        OperationQueue.main.addOperation {
            completion(result)
        }
    }
    task.resume()
}
```

48. Do the same for fetchImage(for:completion:):

```
func fetchImage(for photo: Photo, completion: @escaping (ImageResult) -> Void) {

    let photoURL = photo.remoteURL
    let request = URLRequest(url: photoURL)

    let task = session.dataTask(with: request) {
        (data, response, error) -> Void in

        let result = self.processImageRequest(data: data, error: error)
        OperationQueue.main.addOperation {
            completion(result)
        }
    }
    task.resume()
}
```

49. Build and run the application. Now that the image view is being updated on the main thread, you will have something to show for all your hard work: An image will appear when the web service request finishes. (It might take a little time to show the image if the web service request takes a while to finish.)