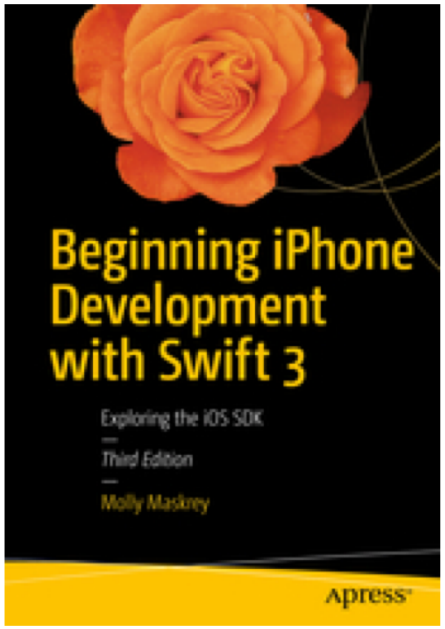
CENTENNIAL COLLEGE



Advanced iOS Development

Week 7

Web Services

Web Services

- You will create an application named **Photorama** that reads in a list of interesting photos from Flickr

- This lesson will lay the foundation and focus on implementing the **web service** requests responsible for fetching the metadata for interesting photos as well as downloading the image data for a specific photo.

Web Services (continued)

- Your web browser uses HTTP to communicate with a web server.
- In the simplest interaction, the browser sends a request to the server specifying a **URL**.
- The server responds by sending back the requested page (typically HTML and images), which the browser formats and displays.

Web Services (continued)

- In more complex interactions, browser requests include other parameters, such as **form data**.
- The server processes these parameters and returns a customized, or dynamic, web page.
- Web browsers are widely used and have been around for a long time, so the technologies surrounding **HTTP** are stable and well developed: **HTTP** traffic passes neatly through most firewalls, web servers are very secure and have great performance, and web application development tools have become easy to use.

Web Services (continued)

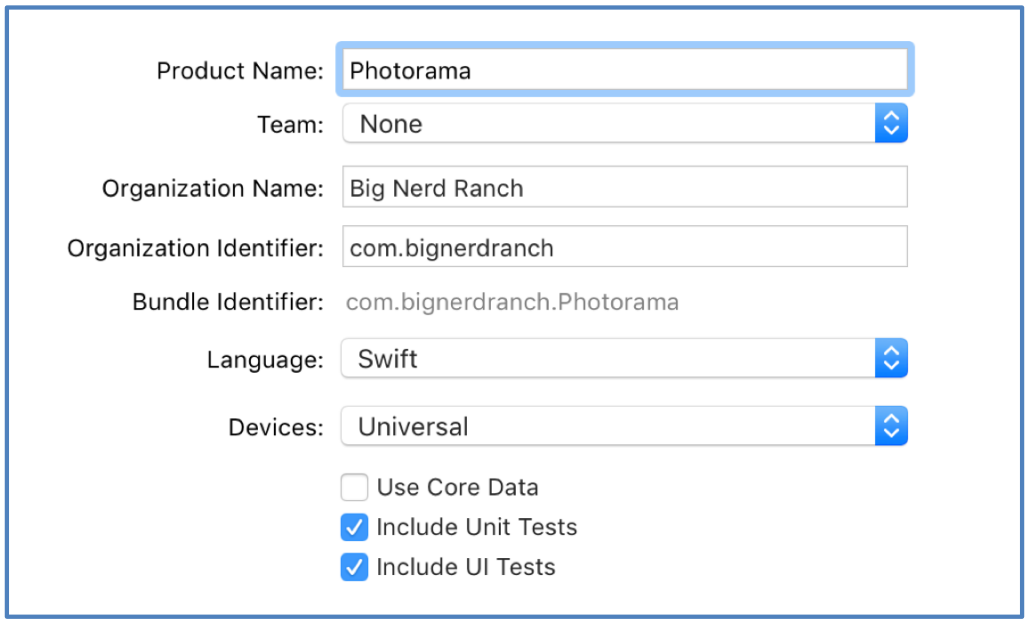
- You can write a client application for iOS that leverages the HTTP infrastructure to talk to a **web-enabled server.**
- The server side of this application is a **web service**.
- Your client application and the web service can exchange requests and responses via HTTP.
- Because HTTP does not care what data it transports, these exchanges can contain complex data.
- This data is typically in JSON (JavaScript Object Notation) or XML format. If you control the web server as well as the client, you can use any format you like.

Web Services (continued)

- **Photorama** will make a web service request to get interesting photos from **Flickr**.
- The web service is hosted at **https://api.flickr.com/services/rest**.
- The data that is returned will be JSON that describes the photos.

Starting the Photorama Application

- Create a new **Single View Application** for the Universal device family.
- Name this application **Photorama**, as shown in the following Figure:



Starting the Photorama Application (continued)

- 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 UIKit

class PhotosViewController: UIViewController {

    @IBOutlet var imageView: UIImageView!

}
```

Starting the Photorama Application (continued)

- In the project navigator, **delete** the existing **`ViewController.swift`**.

- 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`**.

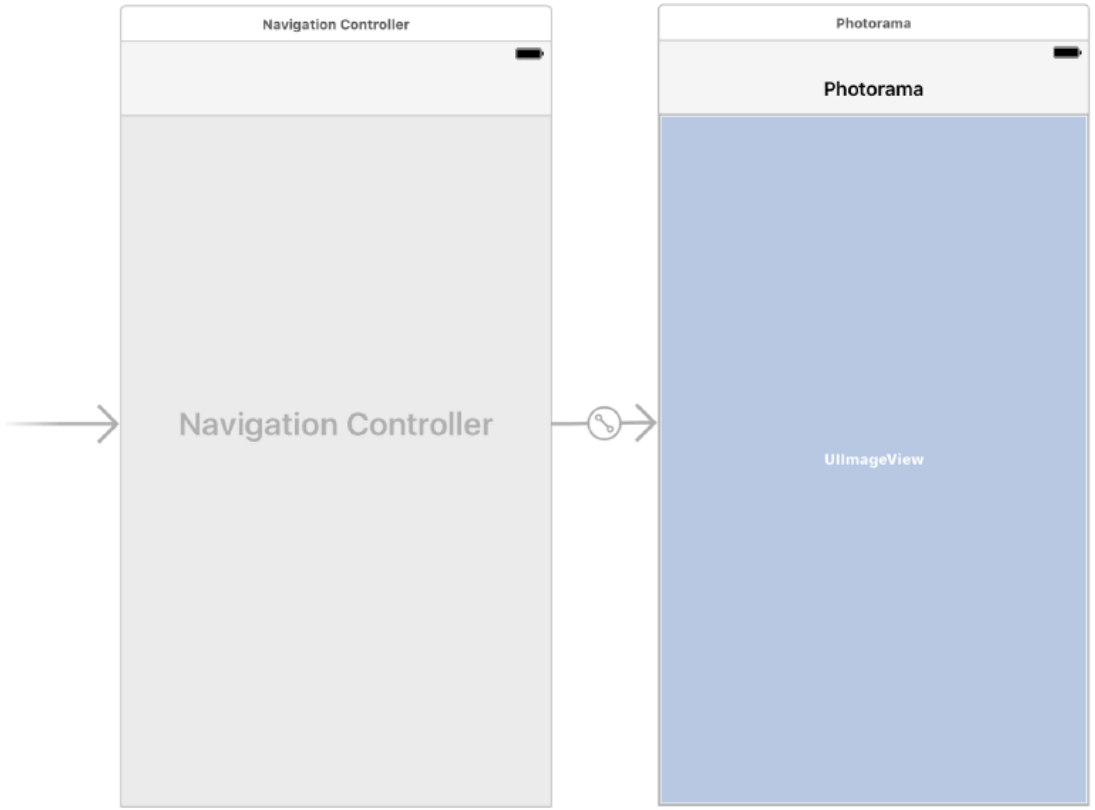
Starting the Photorama Application (continued)

- 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**.
- 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 .

Starting the Photorama Application (continued)

- Finally, double-click on the center of the navigation bar for the **Photos View Controller** and give it a title of "**Photorama.**"

- Your interface will look like:



Building the URL

- Communication with servers is done via **requests** .
- A **request** encapsulates information about the interaction between the application and the server, and its most important piece of information is the destination URL.
- In this section, you will build up the URL for retrieving interesting photos from the Flickr web service.

Building the URL (continued)

- The architecture of the application will reflect best practices.
- For example, each type that you create will encapsulate a single responsibility.
- This will make your types robust and flexible and your application easier to reason about.
- To be a good iOS developer, you not only need to get the job done, but you also need to get it done thoughtfully and with foresight.

Formatting URLs and requests

- The format of a web service request varies depending on the server that the request is reaching out to.
- There are no set-in-stone rules when it comes to **web services**.
- You will need to find the documentation for the **web service** to know how to format a request.
- As long as a client application sends the server what it wants, you have a **working exchange**.

Formatting URLs and requests (continued)

- Flickr's interesting photos web service wants a URL that looks like this:

```
https://api.flickr.com/services/rest/?method=flickr.interestingness.getList
&api_key=a6d819499131071f158fd740860a5a88&extras=url_h,date_taken
&format=json&nojsoncallback=1
```

- Web service requests come in all sorts of formats, depending on what the creator of that web service is trying to accomplish.

- The interesting photos web service, where pieces of information are broken up into **key-value pairs**, is pretty common.

Formatting URLs and requests (continued)

- The **key-value pairs** that are supplied as part of the URL are called **query items** .

- Each of the **query items** for the interesting photos request is defined by and is unique to the Flickr API.

  - The method determines which endpoint you want to hit on the **Flickr** API. For the interesting photos, this is the string "**`flickr.interestingness.getList`**".

  - The **`api_key`** is a key that Flickr generates to authorize an application to use the Flickr API.

Formatting URLs and requests (continued)

- The extras are attributes passed in to customize the response. Here, the **url_h, date_taken** value tells the **Flickr** server that you want the photo URLs to also come back in the response along with the date the photo was taken.
- The format item specifies that you want the payload coming back to be JSON.
- The **`nojsoncallback`** item specifies that you want JSON back in its raw format.

URLComponents

- You will create two types to deal with all of the web service information.

- The `FlickrAPI` struct will be responsible for knowing and handling all Flickr-related information.
- This includes knowing how to generate the URLs that the **Flickr API** expects as well as knowing the format of the incoming JSON and how to parse that JSON into the relevant model objects.
- The `PhotoStore` class will handle the actual web service calls. Let's start by creating the `FlickrAPI struct`.

URLComponents (continued)

- Let's start by creating the **`FlickrAPI struct`**.

- Create a new **Swift** file named **`FlickrAPI`** and declare the **`FlickrAPI`** struct, which will contain all of the knowledge that is specific to the **`Flickr API`**.

```
import Foundation

struct FlickrAPI {

}
```

URLComponents (continued)

- You are going to use an **enumeration** to specify which endpoint on the **Flickr** server to hit.
- For this application, you will only be working with the **endpoint** to get interesting photos.
- However, **Flickr** 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.

URLComponents (continued)

- In **`FlickrAPI.swift`** , create the Method enumeration.
- Each case of Method has a raw value that matches the corresponding **Flickr** endpoint.

```
import Foundation

enum Method: String {
    case interestingPhotos = "flickr.interestingness.getList"
}

struct FlickrAPI {

}
```

- **Enumerations** can have raw values associated with them.
- Although the raw values are often **`Int`**s, you can see here a great use of **String** as the **raw value** for the Method enumeration.

URLComponents (continued)

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

```
enum Method: String {
    case interestingPhotos = "flickr.interestingness.getList"
}

struct FlickrAPI {

    static let baseURLString = "https://api.flickr.com/services/rest"
}
```

- A **type-level property** (or method) is one that is accessed on the type itself – in this case, the **FlickrAPI** type.

- For **structs**, type **properties** and **methods** are declared with the **`static`** keyword; classes use the **`class`** keyword.

- Here, you are declaring a **type-level property** on **FlickrAPI** .

URLComponents (continued)

- The **`baseURLString`** is an implementation detail of the **FlickrAPI** type, and no other type needs to know about it.

- Instead, they will ask for a completed **URL** from **FlickrAPI**.

- To keep other files from being able to access **`baseURLString`**, mark the property as **`private`**.

```
struct FlickrAPI {

    private static let baseURLString = "https://api.flickr.com/services/rest"
}
```

URLComponents (continued)

- This is called **access control**. You can control what can access the properties and methods on your own types. There are **five levels** of **access control** that can be applied to types, properties, and methods:
  - **open** – This is used only for classes, and mostly by framework or third-party library authors. Anything can access this **class**, **property**, or **method**. Additionally, classes marked as open can be subclassed and methods can be **overridden** outside of the module.
  - **public** – This is very similar to open ; however, classes can only be subclassed and methods can only be overridden **inside** (not outside of) the module.
  - **internal** – This is the **default**. Anything in the current module can access this type, property, or method. For an app, only files within your project can access these. If you write a third-party library, then only files within that third-party library can access them – apps that use your third-party library cannot.
  - **fileprivate** – Anything in the same source file can see this type, property, or method.
  - **private** – Anything within the enclosing scope can access this type, property, or method.

URLComponents (continued)

- Now you are going to create a **type method** that builds up the **Flickr** URL for a specific endpoint.

- This method will accept **two arguments**:
  - The first will specify which endpoint to hit using the Method enumeration
  - The second will be an **optional dictionary** of **query item** parameters associated with the request.

URLComponents (continued)

- Implement this method in your **`FlickrAPI`** struct in **FlickrAPI.swift**.
- For now, this method will return an **empty URL**.

CENTENNIAL COLLEGE

```
private static func flickrURL(method: Method,
                              parameters: [String:String]?) -> URL {

    return URL(string: "")!
}
```

URLComponents (continued)

- Notice that the **`flickrURL(method:parameters:)`** method is **`private`**.

- It is an implementation detail of the **FlickrAPI struct**.

- An internal type method will be exposed to the rest of the project for each of the specific endpoint **URLs** (currently, just the interesting photos endpoint).

- These internal type methods will call through to the **`flickrURL(method:parameters:)`** method.

URLComponents (continued)

- In **`FlickrAPI.swift`**, define and implement the **`interestingPhotosURL`** computed property.

```
static var interestingPhotosURL: URL {
    return flickrURL(method: .interestingPhotos,
                     parameters: ["extras": "url_h,date_taken"])
}
```

- Time to construct the full URL. You have the base URL defined as a constant, and the query items are being passed into the **`flickrURL(method:parameters:)`** method via the parameters argument.

- You will build up the URL using the **`URLComponents`** class, which is designed to take in these various components and construct a URL from them.

URLComponents (continued)

- Update the **`flickrURL(method:parameters:)`** method to construct an instance of URLComponents from the base URL.

- Then, loop over the incoming parameters and create the associated **`URLQueryItem`** instances.

```
private static func flickrURL(method: Method,
                              parameters: [String:String]?) -> URL {

    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!
}
```

URLComponents (continued)

- 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 **Flickr** to identify your application and authenticate it with the web service.

- We have generated an API key for this application by creating a **Flickr** account and registering this application. (If you would like your own API key, you will need to register an application at **`www.flickr.com/services/apps/create`** .)

URLComponents (continued)

- In **`FlickrAPI.swift`** , create a constant that references this token.

```
struct FlickrAPI {

    private static let baseURLString = "https://api.flickr.com/services/rest"
    private static let apiKey = "a6d819499131071f158fd740860a5a88"
```

- **Double-check** to make sure you have typed in the **API key** exactly as presented here. It has to match or the server will reject your requests.

URLComponents (continued)

- Finish implementing **`flickrURL(method:parameters:)`** to add the common query items to the **`URLComponents`** .

```
private static func flickrURL(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)
    }

    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!
}
```

Sending the Request

- A **URL request** encapsulates information about the communication from the application to the server.
- Most importantly, it specifies the URL of the server for the request, but it also has a timeout interval, a cache policy, and other metadata about the request. A request is represented by the **`URLRequest`** class.
- Check out the For the More Curious section at the end of this chapter for more information.
- The **`URLSession`** API is a collection of classes that use a request to communicate with a server in a number of ways. The `URLSessionTask` class is responsible for communicating with a server.
- The **`URLSession`** class is responsible for creating tasks that match a given configuration.

Sending the Request (continued)

- In **Photorama**, a new class, **`PhotoStore`** , will be responsible for initiating the web service requests.

- It will use the **URLSession API** and the **FlickrAPI** struct to fetch a list of interesting 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 {

}
```

URLSession

- Let's look at a few of the properties on **`URLRequest`** :
  - **`allHTTPHeaderFields`** – a dictionary of metadata about the HTTP transaction, including character encoding and how the server should handle caching
  - **`allowsCellularAccess`** – a Boolean that represents whether a request is allowed to use cellular data
  - **`cachePolicy`** – the property that determines whether and how the local cache should be used
  - **`httpMethod`** – the request method; the default is GET, and other values are POST, PUT, and DELETE
  - **`timeoutInterval`** – the maximum duration a connection to the server will be attempted for

URLSession (continued)

- In **PhotoStore.swift** , add a property to hold on to an instance of `URLSession` .

CENTENNIAL COLLEGE

```
class PhotoStore {

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

}
```

URLSession (continued)

- In **`PhotoStore.swift`** , implement the **`fetchInterestingPhotos()`** method to create a URLRequest that connects to **api.flickr.com** and asks for the list of interesting photos.

- Then, use the **`URLSession`** to create a **`URLSessionDataTask`** that transfers this request to the server.

URLSession (continued)

CENTENNIAL COLLEGE

```
func fetchInterestingPhotos() {

    let url = FlickrAPI.interestingPhotosURL
    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 interesting photos: \(requestError)")
        } else {
            print("Unexpected error with the request")
        }
    }
    task.resume()
}
```

URLSession (continued)

- 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!
```

URLSession (continued)

- The store is a dependency of the **PhotosViewController** .

- Open **`AppDelegate.swift`** and use property injection to give the **PhotosViewController** an instance of **PhotoStore** .

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
        launchOptions: [UIApplicationLaunchOptionsKey : Any]?) -> Bool {
    // Override point for customization after application launch.

    let rootViewController = window!.rootViewController as! UINavigationController
    let photosViewController =
        rootViewController.topViewController as! PhotosViewController
    photosViewController.store = PhotoStore()

    return true
}
```

URLSession (continued)

- 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 interesting photos.

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

    store.fetchInterestingPhotos()
}
```

Modeling the Photo

- 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.

Modeling the Photo (continued)

CENTENNIAL COLLEGE

```
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
    }
}
```

JSONSerialization

- 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.

JSONSerialization (continued)

- Open **`PhotoStore.swift`** and update **`fetchInterestingPhotos()`** to print the JSON object to the console.

CENTENNIAL COLLEGE

```
func fetchInterestingPhotos() {

    let url = FlickrAPI.interestingPhotosURL
    let request = URLRequest(url: url)
    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 interesting photos: \(requestError)")
        } else {
            print("Unexpected error with the request")
        }
    }
    task.resume()
}
```

Parsing JSON data

- 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 {
```

Parsing JSON data (continued)

- In **`FlickrAPI.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.

CENTENNIAL COLLEGE

```
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)
    }
}
```

Parsing JSON data (continued)

- At the top of **`FlickrAPI.swift`** , declare a custom enum to represent possible errors for the Flickr API.

CENTENNIAL COLLEGE

```
enum FlickrError: Error {
    case invalidJSONData
}

enum Method: String {
    case interestingPhotos = "flickr.interestingness.getList"
}
```

Parsing JSON data (continued)

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

CENTENNIAL COLLEGE

```
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(FlickrError.invalidJSONData)
        }

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

Parsing JSON data (continued)

- In FlickrAPI.swift , add a constant instance of **`DateFormatter`** .

```
private static let baseURLString = "https://api.flickr.com/services/rest"
private static let apiKey = "a6d819499131071f158fd740860a5a88"

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

Parsing JSON data (continued)

- Still in **`FlickrAPI.swift`** , write a new method to parse a JSON dictionary into a **Photo** instance.

CENTENNIAL COLLEGE

```
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)
}
```

Parsing JSON data (continued)

- Now update **`photos(fromJSON:)`** to parse the dictionaries into Photo instances and then return these as part of the success enumerator.

Parsing JSON data (continued)

CENTENNIAL COLLEGE

```
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(FlickrError.invalidJSONData)
        }

        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(FlickrError.invalidJSONData)
        }
        return .success(finalPhotos)
    } catch let error {
        return .failure(error)
    }
}
```

Parsing JSON data (continued)

- 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 FlickrAPI.photos(fromJSON: jsonData)
}
```

Parsing JSON data (continued)

- Now, update **`fetchInterestingPhotos()`** to use the method you just created.

CENTENNIAL COLLEGE

```
func fetchInterestingPhotos() {

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

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

Parsing JSON data (continued)

- Finally, update the method signature for **`fetchInterestingPhotos()`** to take in a completion closure that will be called once the web service request is completed.

CENTENNIAL COLLEGE

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

    let url = FlickrAPI.interestingPhotosURL
    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()
}
```

Parsing JSON data (continued)

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

CENTENNIAL COLLEGE

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

    store.fetchInterestingPhotos() {
        (photosResult) -> Void in

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

    }
}
```

Downloading and Displaying the Image 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 UIKit

enum ImageResult {
    case success(UIImage)
    case failure(Error)
}

enum PhotoError: Error {
    case imageCreationError
}

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

Downloading and Displaying the Image Data (continued)

- Now, in the same file, implement a method to download the image data. Like the **`fetchInterestingPhotos(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()
}
```

Downloading and Displaying the Image Data (continued)

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

CENTENNIAL COLLEGE

```
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)
}
```

Downloading and Displaying the Image Data (continued)

- 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()
}
```

Downloading and Displaying the Image Data (continued)

- To test this code, you will download the image data for the first photo that is returned from the interesting 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)")
        }
    }
}
```

Downloading and Displaying the Image Data (continued)

- Now update **`viewDidLoad()`** to use this new method.


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

    store.fetchInterestingPhotos {
        (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 interesting photos: \(error)")
        }
    }
}
```

The Main Thread

- In PhotoStore.swift , update **`fetchInterestingPhotos(completion:)`** to call the completion closure on the main thread.

CENTENNIAL COLLEGE

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

    let url = FlickrAPI.interestingPhotosURL
    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()
}
```

The Main Thread (continued)

- 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()
}
```