The right way to use PHPicker and retrieve EXIF data without requesting library permissions in Swift

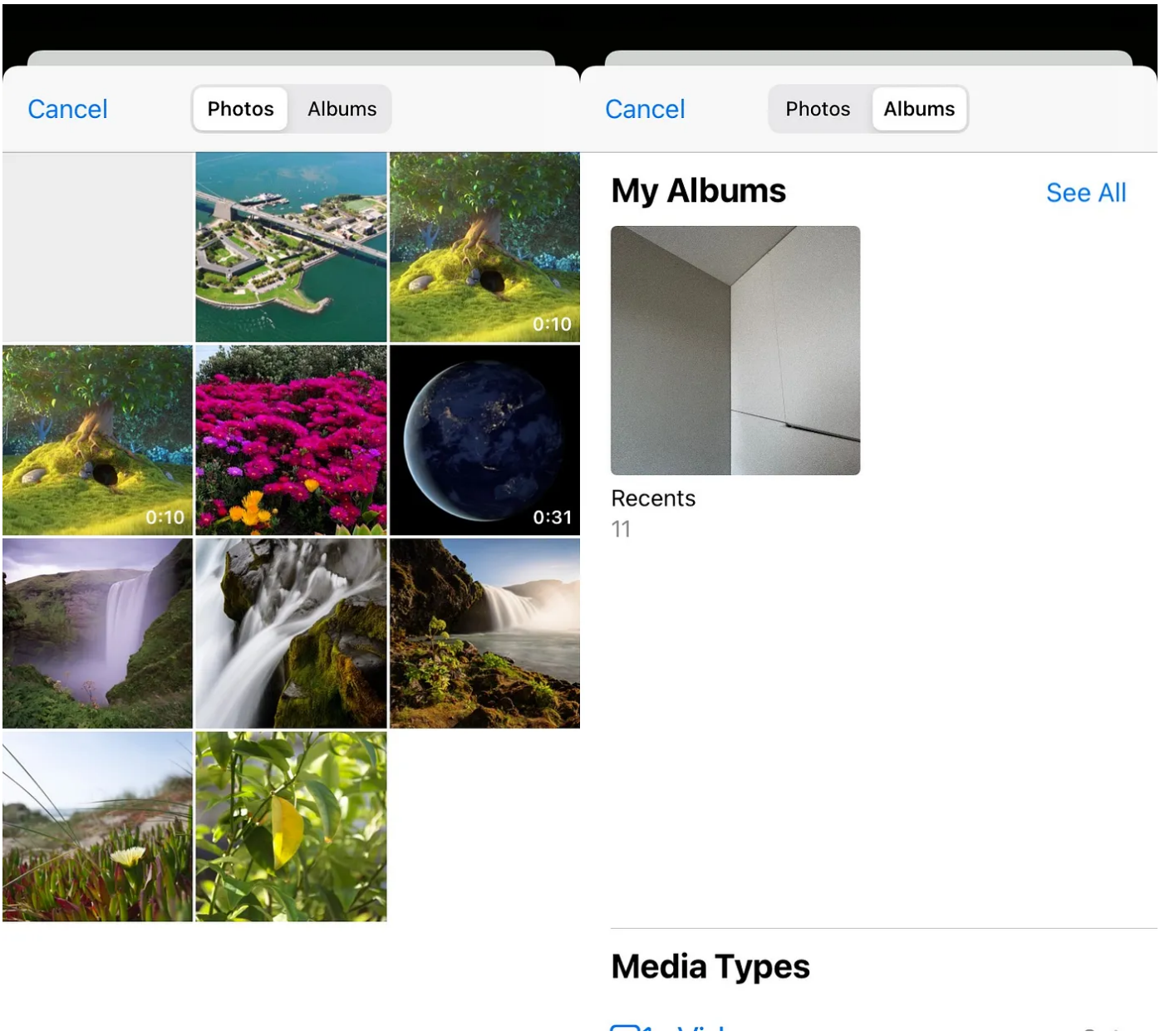
Luke Brandon Farrell · Follow
Published in ITNEXT
7 min read · Jun 8, 2022

In this short article, I will outline the correct way to use PHPicker and when not to use PHPicker. The reasoning behind this article is due to the frustration I encountered when trying to build out a native library with PHPicker. Many articles across the internet are limited in their approach and do not address some of the core issues and frustrations with PHPicker and iOS permissions.

Let's talk about the problems.

What is PHPicker?

As of iOS 14, PHPicker is the system-provided Picker, which allows you to get access to photos and videos from the users photo library [1].



Problems with PHPicker and Library Permissions

> Library permissions are not supposed to be used with PHPicker.

There are some fundamental user experience problems when using PHPicker with library permissions. But first, why would we want to request library permissions when using PHPicker?

The top ranked article on the internet encourages you to use PHAsset with PHPicker to fetch extra data such as exif and location information for the result [2]. The problem with this is that to fetch a PHAsset you must have library permission to access the asset

which goes against the core philosophy of PHPicker: a picker which can be used without needing to request permissions.

Apple themselves even suggest that you should request permissions only when "neccesary" but do not go into detail about the edge cases this causes [3].

Like I said... hours of frustration where all the tutorials I encountered encouraged you to use PHAsset to fetch metadata. None suggested other ways you could get exif metadata, which is what I will provide you with today, other ways... the correct way.

Alongside needing to request permissions, another issue is when the user chooses to select limited library access. This can create some confusing user experience issues. PHPicker will still show all photos (even photos not granted permission via limited picker) and selecting a photo which wasn't given permissions via the limited photo selection dialog will result in the app not being able to retrieve the PHAsset [4]. Without the PHAsset, most tutorial on how to fetch metadata and handle the result of PHPicker fall short.

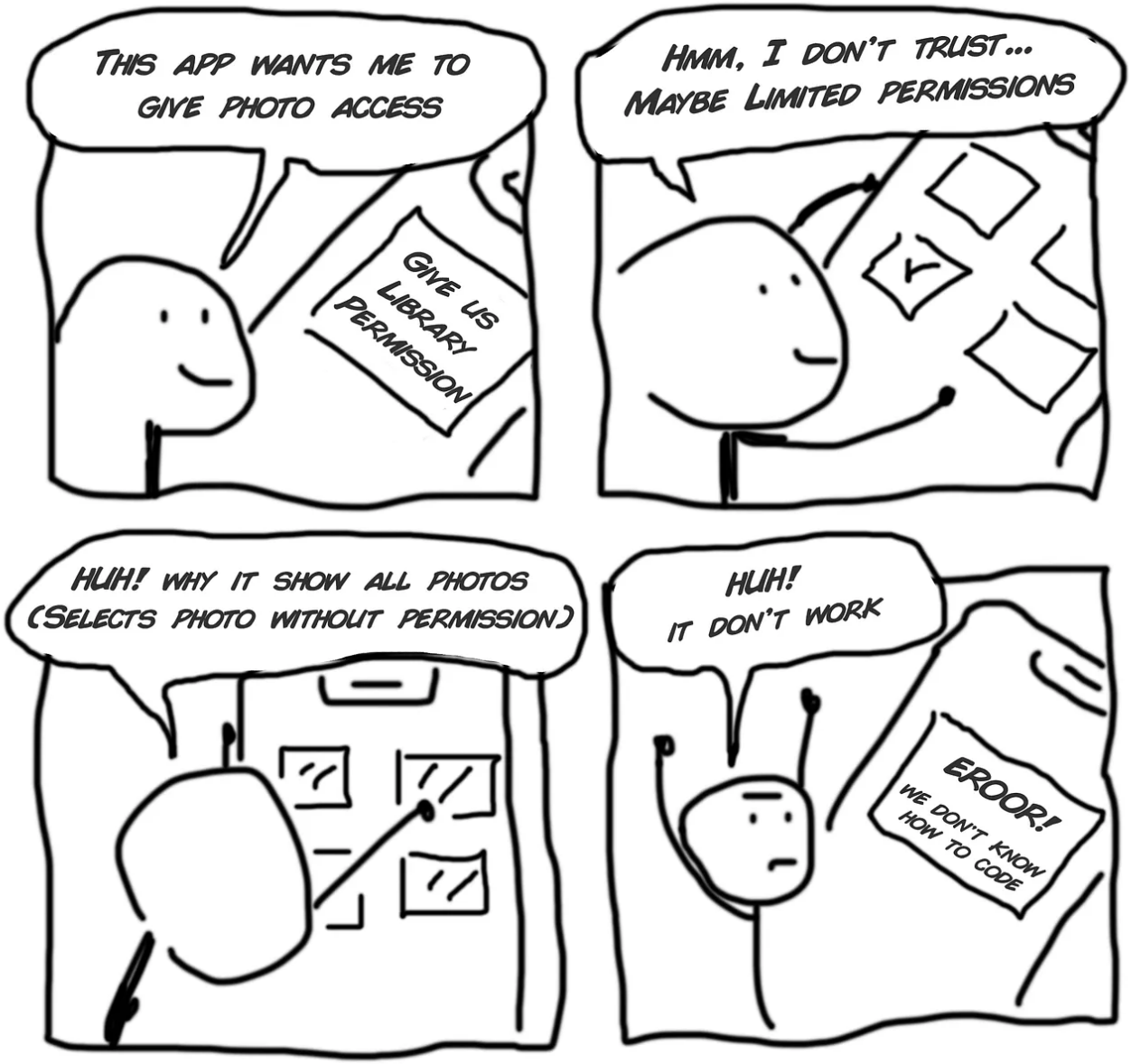
From the users standpoint, it looks something like this:

Your application requests permissions to access the user library
User says: "I shall only give this application limited access to some photos"

Your application opens PHPicker and displays all photos
User says: "Huh, I thought I only give limited access, oh well"

User selects a photo which they haven't given us access to.
Application now needs to either do nothing or explain to the user that to retrieve certain metadata for the photo they selected they must yet again update their permissions.

User is confused.



Comic Strip about PHPicker

All these issues occur because we are using library permissions combined with PHPicker. PHPicker was never made to be used that way. Many tutorials recommend this way, inadvertently by suggesting using PHAsset as a straightforward way of extracting exif data when using PHPicker. Using PHAsset with PHPicker is an anti-pattern.

The Right Way to use PHPicker

For the sake of creating an easy to follow guide, let's start from the beginning.
You want to setup your PHPicker with a configuration and present it:

```
var configuration = PHPickerConfiguration();
...

let picker = PHPickerViewController(configuration: configuration);
picker.delegate = self
self.present(picker, animated: true, completion: nil)
```

Then make sure your class extends both *PHPickerViewControllerDelegate*, *UIAdaptivePresentationControllerDelegate* and implements the *picker()* instance method:

```
class MyPickerClass: PHPickerViewControllerDelegate,
UIAdaptivePresentationControllerDelegate {

   func picker(
     _ picker: PHPickerViewController,
     didFinishPicking results: [PHPickerResult]
   ) {
      // This is where the magic happens 🪄
   }
}
```

Now in the *picker* method this is where we can retrieve the asset and metadata. At this point most articles either instruct you to use *NSItemProvider* (which doesn't need library permissions) but none tell you how to get the exif data after that or they tell you to use *PHAsset* which is the incorrect way to use *PHPicker* as we'd need permissions [5].

> PHAsset should not be used with PHPicker, it's not the correct way of using these APIs

Now let's get to the code.

Firstly we'd want to loop through the result and extract the NSItemProvider:

```
func picker(
  _ picker: PHPickerViewController,
  didFinishPicking results: [PHPickerResult]
) {
  for result in results {
     let itemProvider: NSItemProvider = result.itemProvider;
  }
}
```

Using the *itemProvider* we can now read the type of object and handle it based on if it's a photo, video or something else. There are a few ways of doing this but the best I have found are:

```
func picker(_ picker: PHPickerViewController, didFinishPicking
results: [PHPickerResult]) {

for result in results {
        let itemProvider: NSItemProvider = result.itemProvider;

if(itemProvider.hasItemConformingToTypeIdentifier(UTType.image.identif
ier)) {
   // Handle the image here
        } else
if(itemProvider.hasItemConformingToTypeIdentifier(UTType.movie.identif
ier)) {
          // Handle the video here
        } else {
          // Unsupported asset
        }
    }
}
```

Once we know the type of the asset we can load it. The best way to load an asset from a *NSItemProvider* which allows us to easily access exif data in this case is *loadFileRepresentation*. Although you can use *loadItem*, *loadFileRepresentation*, *loadObject* etc depedning on your requirements. See NSItemProvider docs.

```
func picker(_ picker: PHPickerViewController, didFinishPicking
results: [PHPickerResult]) {

for result in results {
        let itemProvider: NSItemProvider = result.itemProvider;
```

```
if(itemProvider.hasItemConformingToTypeIdentifier(UTType.image.identif
ier)) {
   itemProvider.loadFileRepresentation(forTypeIdentifier:
UTType.image.identifier) { data, error in
    // Now we can get EXIF data
          }
       }
```


loadFileRepresentation on an image. We can use that to load the exif data from the asset:

```
if let url = data as? URL {
   let options = [kCGImageSourceShouldCache as String:
kCFBooleanFalse]
   let data = NSData(contentsOf: url)
   let imgSrc = CGImageSourceCreateWithData(data, options as
CFDictionary)
   let metadata = CGImageSourceCopyPropertiesAtIndex(imgSrc, 0,
options as CFDictionary)
}
```

The metadata data object looks like this (copied from Xcode console):

```
ColorModel = RGB;
DPIHeight = 72;
DPIWidth = 72;
Depth = 8;
Orientation = 6;
PixelHeight = 3024;
PixelWidth = 4032;
PrimaryImage = 1;
ProfileName = "Display P3";
"{TIFF}" = {
    DateTime = "2022:05:26 14:28:29";
    HostComputer = "iPhone XR";
    Make = Apple;
    Model = "iPhone XR";
    Orientation = 6;
    ResolutionUnit = 2;
    Software = "15.4.1";
    TileLength = 512;
    TileWidth = 512;
```

```
        XResolution = 72;
        YResolution = 72;
    },
    "{Exif}" = {
        ApertureValue = "1.69599381283836";
        BrightnessValue = "2.126521070833867";
        ColorSpace = 65535;
        CompositeImage = 2;
        DateTimeDigitized = "2022:05:26 14:28:29";
        DateTimeOriginal = "2022:05:26 14:28:29";
        ExifVersion =     (
            2,
            3,
            2
        );
        ExposureBiasValue = 0;
        ExposureMode = 0;
        ExposureProgram = 2;
        ExposureTime = "0.02";
        FNumber = "1.8";
        Flash = 16;
        FocalLenIn35mmFilm = 26;
        FocalLength = "4.25";
        ISOSpeedRatings =     (
            200
        );
        LensMake = Apple;
        LensModel = "iPhone XR back camera 4.25mm f/1.8";
        LensSpecification =     (
            "4.25",
            "4.25",
            "1.8",
            "1.8"
        );
        MeteringMode = 5;
        OffsetTime = "+02:00";
        OffsetTimeDigitized = "+02:00";
        OffsetTimeOriginal = "+02:00";
        PixelXDimension = 4032;
        PixelYDimension = 3024;
        SceneType = 1;
        SensingMethod = 2;
        ShutterSpeedValue = "5.644289064920122";
        SubjectArea =     (
            2013,
            1511,
            2217,
            1330
        );
        SubsecTimeDigitized = 579;
        SubsecTimeOriginal = 579;
```

```
        WhiteBalance = 0;
    },
    "{GPS}" = {
        Altitude = "19.96612929490311";
        AltitudeRef = 0;
        DestBearing = "272.5662841313811";
        DestBearingRef = T;
        HPositioningError = "7.460463890353189";
        ImgDirection = "272.5662841313811";
        ImgDirectionRef = T;
        Latitude = "39.47537";
        LatitudeRef = N;
        Longitude = "0.3831666666666667";
        LongitudeRef = W;
        Speed = 0;
        SpeedRef = K;
    },
```

Then from the metadata variable you can extract properties as such:

```
metadata[kCGImagePropertyColorModel];
metadata[kCGImagePropertyPixelWidth];
metadata[kCGImagePropertyPixelHeight];
```

It can be hard to work with this object so I have created a class called ExifData which you can use to work with the object, it currently supports twenty-six (26) values from the exif object, you can expand the class to your needs. It can be used as such:

```
let exif = ExifData(data: data);
let exif = ExifData(url: url);

exif.dateTimeDigitized
```

It supports being initialised with *Data*, *URL* or *UIImage* meaning you can use the following methods *loadItem* (will return a *UIImage*), *loadDataRepresentation* (will return a *Data* object) and *loadFileRepresentation* (will return a *URL*).

IMPORTANT! One thing to note is that *UIImage.pngData()* and *UIImage.jpegData()* strip the exif data from the resulting *Data* object returned by these methods. So if you use

these methods for compressions, make sure to extract the exif data before!

There we have it! We have a way to get an asset using *PHPicker* and retrieve the exif data without requesting permissions. Due to the *ExifData* class I have shared with you accepting *URL*, *Data* and *UIImage* data types it should give you the flexibility to do what other operations you may want to perform on the image after selection, such as compression and resizing. Here is a great article on resizing and compressing of a UIImage: Image Resizing Techniques — NSHipster.

References

- [1] — Apple Developer Documentation [Apple Developer Documentation] (https://developer.apple.com/documentation/photokit/phpickerviewcontroller).
- [2] — Top Ranked Article suggesting the use of PHAsset with PHPicker: [Get photo metadata with PHPicker in SwiftUI — Felix Larsen] (https://www.felixlarsen.com/blog/photo-metadata-phpickerview).
- [3] — Apple Suggesting Permissions should only be requested if “necessary” [Meet the new Photos picker — WWDC20 — Videos — Apple Developer] (https://developer.apple.com/videos/play/wwdc2020/10652/?time=633).
- [4] — PHPicker and Permissions Confusion [Confused about PHPicker’s privacy | Apple Developer Forums](https://developer.apple.com/forums/thread/652819).
- [5] — An In Depth Guide on PHPicker (without covering how to extract exif data) [The Complete Guide of PHPicker in iOS 14](https://www.appcoda.com/phpicker/).

Swift | Phpickerviewcontroller | Phasset | Uiimage | Exif Data

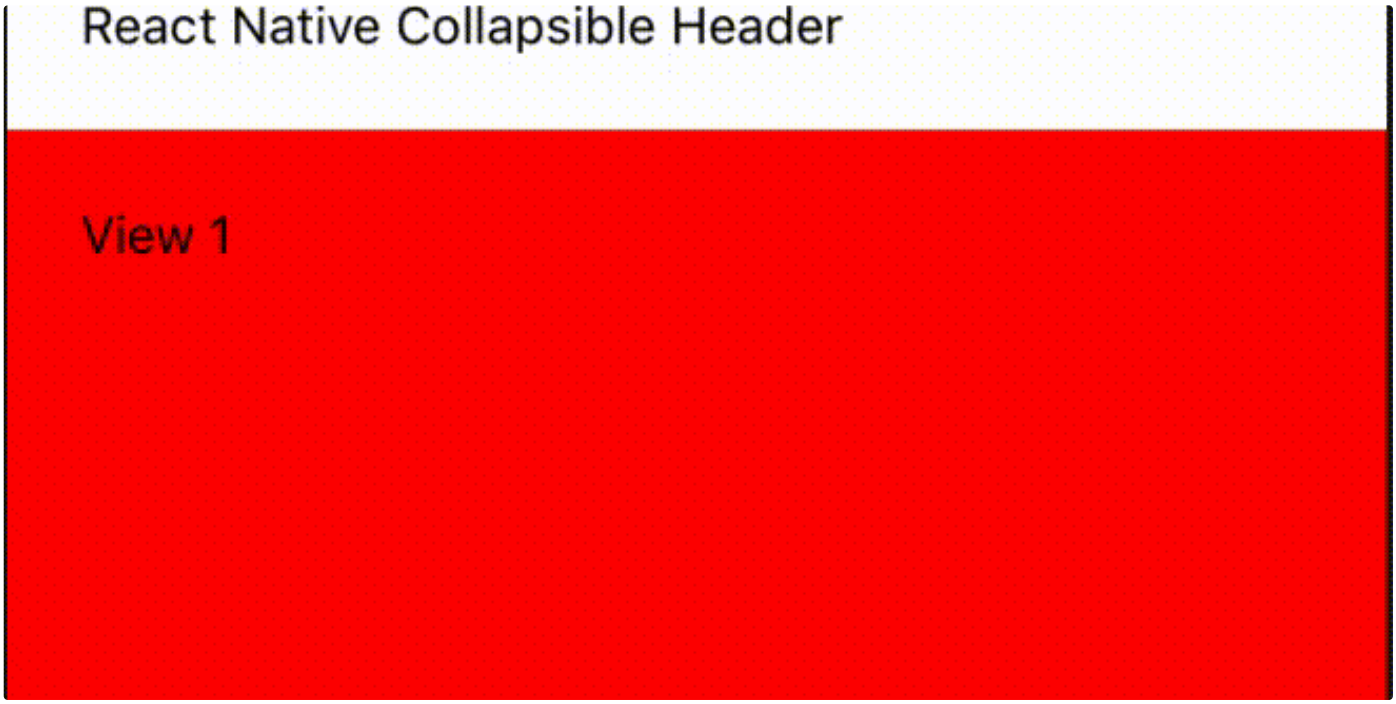
Follow

Written by Luke Brandon Farrell

75 Followers · Writer for ITNEXT

Luke is using Swift, Java and React Native to build epic mobile applications. He writes about people, code, architecture and business ventures.

More from Luke Brandon Farrell and ITNEXT



 Luke Brandon Farrell in ITNEXT

React Native Collapsible Headers—Expert Knowledge Revealed

Quickly make sizeable improvements to the quality of your apps. The recipe 🥙 step-by-step behind collapsable headers in React Native.

3 min read · Oct 12, 2019

166 3

Carlos Arguelles in ITNEXT

Amazon's Not So Secret Weapon

The magic of Working Backwards: a real-world case study

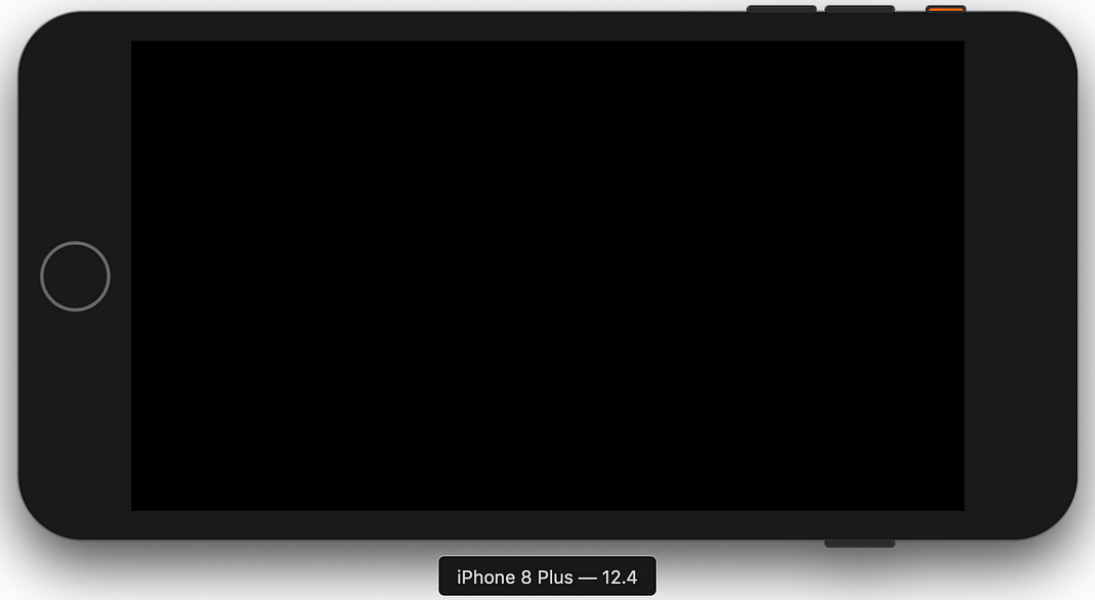
10 min read · Aug 7, 2022

Vitalii Shevchuk in ITNEXT

🦙 How to Run Llama 2 on Mac M1 and Train with Your Own Data

Llama 2 is the next generation of large language model (LLM) developed and released by Meta, a leading AI research company. It is...

✨ · 6 min read · Jul 28



Luke Brandon Farrell in Qeepsake Engineering

Lock Device Orientation in React Native Apps

Lock device orientation in React Native 🔒 The different methods for locking orientation on iOS, Android, Expo and React Native Navigation.

2 min read · Oct 11, 2019

14 1

See all from Luke Brandon Farrell

See all from ITNEXT

Recommended from Medium

Sarim Khan

SwiftUI Permissions

In iOS development, permissions refer to the user's explicit grant of access to certain resources or features on their device, such as the...

4 min read · Apr 26

9

Janvi Arora

URLSession in Swift

In Swift, URLSession is a powerful framework for networking that allows developers to send and receive data from the web. URLSession...

4 min read · Mar 3

 1

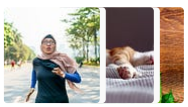
Lists

Staff Picks

415 stories · 236 saves

Stories to Help You Level-Up at Work

19 stories · 184 saves

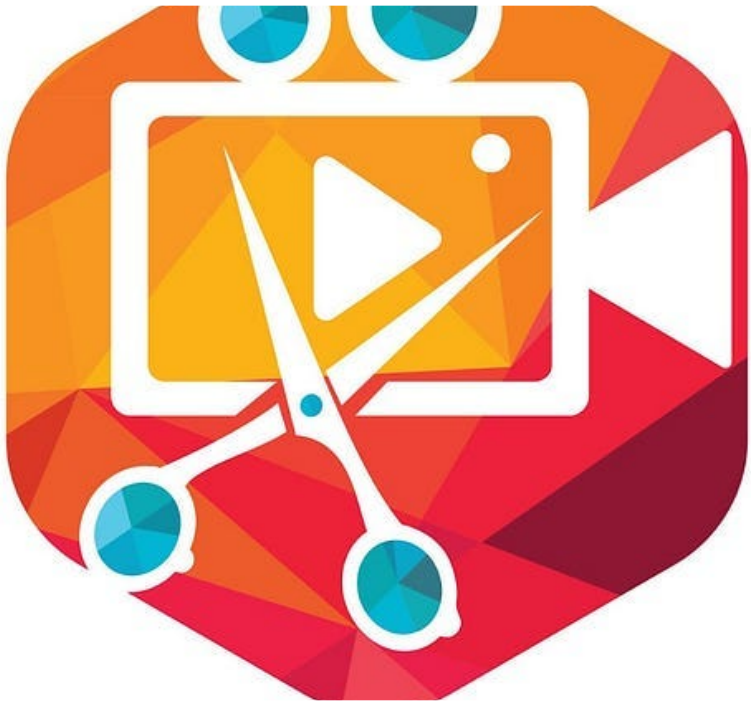


Self-Improvement 101

20 stories · 457 saves

Productivity 101

20 stories · 453 saves



CaratLane Insider in CaratLane Insider

Video Editing in iOS using AV Foundation

This article will give you a fair understanding of video editing using AV foundation.

7 min read · Mar 10

Unbecoming

10 Seconds That Ended My 20 Year Marriage

It's August in Northern Virginia, hot and humid. I still haven't showered from my morning trail run. I'm wearing my stay-at-home mom...

· 4 min read · Feb 16, 2022

899

Ckinet Andrii

How to add Lottie Animation in SwiftUI

Using Lottie, we can bring more dynamism to our apps, which will result in an improved user experience.

3 min read · Feb 28

2