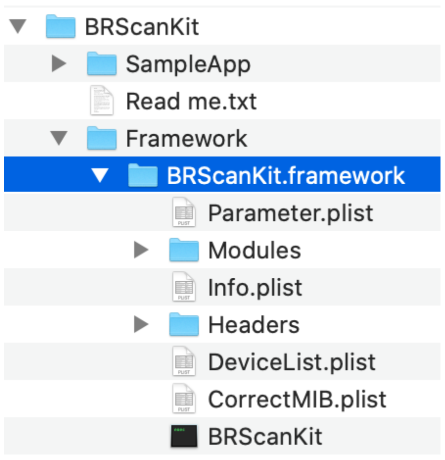
BRScanKit Swift Documentation

Installation Into a New Project

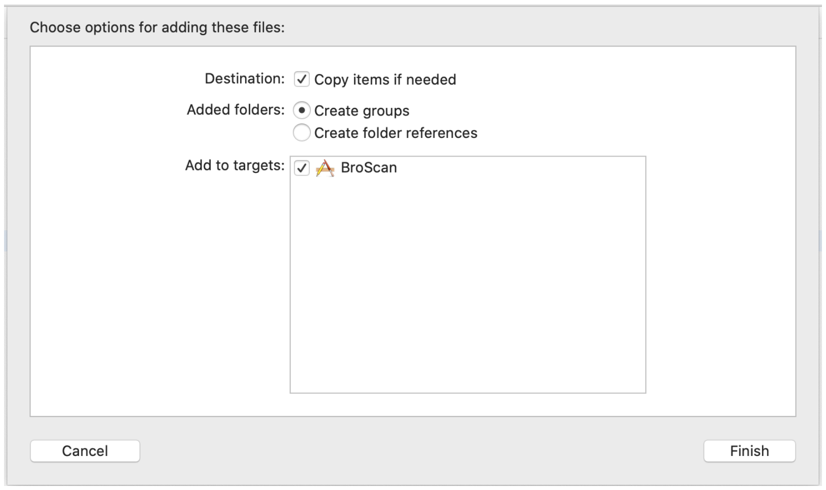
Locate the BRScanKit.framework directory in the BRScanKit project files. The directory should be inside of the Framework directory as shown here.



Take a moment to explore the SampleApp and Documents directories. There you will find an Objective-C scanning application and HTML formatted documentation for the framework.

To install the SDK into your project, drag the BRScanKit.framework directory from a finder window and drop it into Project Navigator pane. If it seems like the framework files won't "stick" to the navigator hierarchy, try placing it further up the tree view. The trick is to expose a horizontal insertion cursor just before dropping the directory.

Make sure that the **Copy items if needed** checkbox is active. Select **Create groups** and double check that your application targets are correctly set.

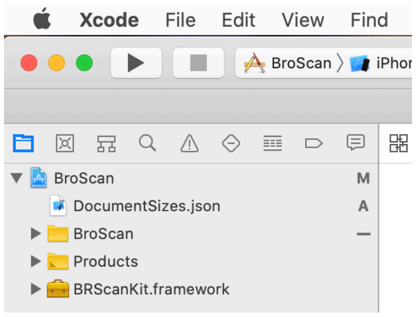


Choose options for adding these files:

Destination: ✓ Copy items if needed

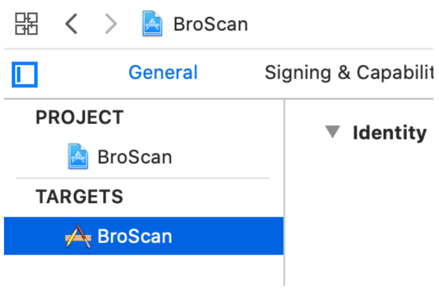
Added folders: ● Create groups
○ Create folder references

Add to targets: ✓ BroScan

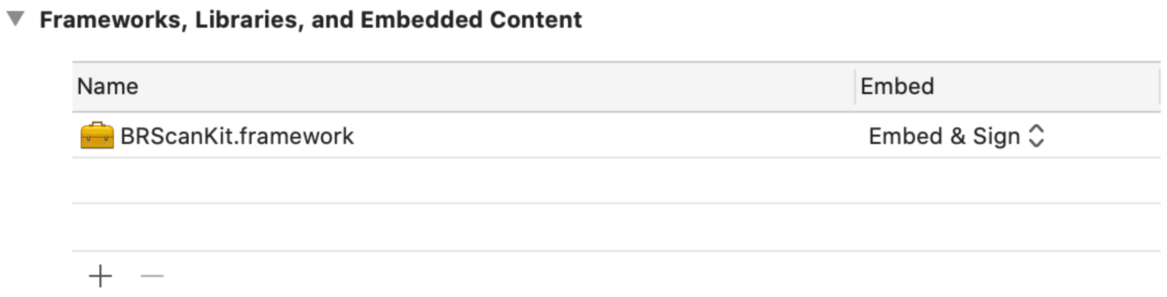


Configure the Framework Asset

In the Project Navigator, select your project parent node. This will display your project settings and allow you to embed the BRScanKit framework into your app. Make sure that the **General** page tab is selected and click on your project's app target.



Find the **Frameworks, Libraries, and Embedded Content** pane. Switch the **Embed** option to **Embed & Sign**.



Frameworks, Libraries, and Embedded Content

Name	Embed
BRScanKit.framework	Embed & Sign

Working with the Device Browser

The BRScanKit framework includes a device browser that can search your network for compatible Brother scanners. It uses Apple's Bonjour zero-configuration discovery service.

To use the browser, import the BRScanKit into your class file:

```
import BRScanKit
```

You will need an instance of the Brother Device Browser and an array to hold all of the devices found by the service:

```
let brotherBrowser = BRScanDeviceBrowser()
var brotherDevices = [BRScanDevice]()
```

To control the service, use the **search()** and **stop()** methods:

```
brotherBrowser.search()
brotherBrowser.stop()
```

The service requires delegated methods to handle the devices it finds. Assign the delegate to the browser (usually self):

```
brotherBrowser.delegate = self
```

Supply the browser with the delegated methods **scanDeviceBrowser(_:didFindDevice:)** and **scanDeviceBrowser(_:didRemoveDevice:)**. Discovered devices are passed to the scanDeviceBrowser(_:didFindDevice:) method for handling by you.

Delegate Method	Parameters
scanDeviceBrowser	_ browser: BRScanDeviceBrowser! // Delegator didFindDevice device: BRScanDevice! // A scanner found by the browser
scanDeviceBrowser	_ browser: BRScanDeviceBrowser! // Delegator didRemoveDevice device: BRScanDevice! // A removed scanner

An example is detailed below:

```
extension ViewController: BRScanDeviceBrowserDelegate {

      func scanDeviceBrowser(_ browser: BRScanDeviceBrowser!, didFind device:
            BRScanDevice!) {

            brotherDevices.append(device)
      }

      func scanDeviceBrowser(_ browser: BRScanDeviceBrowser!, didRemove
            device: BRScanDevice!) {

            guard let index = brotherDevices.firstIndex(of: device)
                  else { return }

            brotherDevices.remove(at: index)
      }

}
```

Note: with this version of the SDK, it seems like the browser does not keep track of device objects during a search session. Devices that fall out of network are not tracked and delegated to the scanDeviceBrowser(_:didRemove) method. If you were to call search() again, new device objects for the same scanners will be created. If you want to search again for devices as they come in and out of the network, empty the device array and then call search().

The BRScanDeviceBrowser object can be used to simulate device searches. Key/value pairs are used to configure the simulator. The dictionary is described below:

Property	Description
options	Dictionary used to set Virtual Search Mode parameters

Keys and values for the **options** dictionary are detailed below. To use the dictionary, just pass the key name—it is a string property holding the actual key for the dictionary.

Key Name	Value Type	Description
BRScanDeviceBrowserOptionUseVirtualSearchModeKey	Bool	Set the value to true if you want to simulate a device search
BRScanDeviceBrowserOptionVirtualSearchDeviceCountKey	Int	Specify an integer 1 and 999 for the number of devices to generate
BRScanDeviceBrowserOptionVirtualSearchDeviceIntervalKey	CGFloat	The time in seconds between discovered virtual devices

Examining a Scanner

The BRScanDevice returned from the browser contains properties about the scanner that are useful and necessary when configuring a scanning job.

Property	Description
modelName	Model name and number of the device
deviceURI	Bonjour device URI
ipAddress	IP address of the device
capability	Dictionary of machine capabilities for the scanner model

Keys and value types for the **capability** property are detailed below. To use the dictionary, just pass the key name—it is a string property holding the actual key for the dictionary.

Key	Value Type	Description
BRScanDeviceCapabilityIsScannerAvailableKey	Bool	Reports if the scanner is available for use
BRScanDeviceCapabilityIsColorScannerAvailableKey	Bool	Color scanning availablility
BRScanDeviceCapabilityIsFBScannerAvailableKey	Bool	Flatbed scanning availablility
BRScanDeviceCapabilityIsADFScannerAvailableKey	Bool	Automatic document feeder availability
BRScanDeviceCapabilityIsDuplexScannerAvailableKey	Bool	Duplexing availability
BRScanDeviceCapabilityIsAutoDocumentSizeAvailableKey	Bool	Document size sensing capability
BRScanDeviceCapabilityMaxScanDocumentKey	*enum	Maximum document size limit of the scanner
BRScanDeviceCapabilityMaxDuplexScanDocumentKey	*enum	Maximum document size limit of the scanners duplexer

*enum **BRScanDeviceCapabilityMaxScanDocument**
- BRScanDeviceCapabilityMaxScanDocumentA3
- BRScanDeviceCapabilityMaxScanDocumentLegal
- BRScanDeviceCapabilityMaxScanDocumentA4

Use the results of the **capability** dictionary to learn what functions your scanner can perform before submitting a scanning job.

Here is a practical example of how a BRScanDevice instance is used:

```
if let device = brotherDevices.first {
      let isAvailable = device.capability[
            BRScanDeviceCapabilityIsScannerAvailableKey]
      let isDuplexAvailable = device.capability[
            BRScanDeviceCapabilityIsDuplexScannerAvailableKey]
      let ip = device.ipAddress
}
```

BRScanDevice contains several class methods that can initialize scanners, but it is much easier to use the results from BRScanDeviceBrowser.

Class Method	Description
BRScanDevice.init(ipAddress: String!)	Initializes a device with the supplied IP
BRScanDevice.init(modelName: String!, deviceURI: String!, ipAddress: String!)	Initializes a device with the supplied model name, URI and IP
BRScanDevice.resolveIPAddress(fromURI: String!)	Returns an IP for the supplied URI

Building and Running a Scan Job

Before a scan can be performed, the scanner needs job requirements specified in an instance of **BRScanJob**. This object contains all of the job details required to produce scanned images.

Begin by creating an instance of **BRScanJob** (in this case, using the first scanner found in our BRScanDevice array from an early example above):

```
let selectedDevice = brotherDevices[0]
let scanJob = BRScanJob(ipAddress: selectedDevice.ipAddress)
```

The BRScanJob **init(ipAddress: String!)** initializer conveniently assigns a targeted scanner using an IP address. You may also create an instance of the job object without the address and assign it later, but before submitting the job for processing.

The **start()** and **cancel()** methods control job processing:

```
scanJob.start()
scanJob.cancel()
```

Like the browser, this service requires delegated methods to manage scanning results. Assign the delegate to the job service:

```
scanJob.delegate = self
```

This service requires that you build delegated methods **scanJob(_:progress:), scanJob(_:didFinishPage:)** and **scanJobDidFinish(_:result:)**. scanJob(_:progress:) can be used to periodically update progress with a scanning job. When the progress value is tied to a progress indicator, multiple page jobs may cause the indicator to run backwards during processing updates. The scanJob(_:didFinishPage:) method will provide you with a file path to a scanned image. The scanJobDidFinsh(_:result:) is used to handle all of your end of job tasks.

Delegate Method	Parameters
scanJob	_ job: BRScanJob \\ Delegator progress: Float \\ Progress indicator value
scanJob	_ job: BRScanJob \\ Delegator didFinishPage path: String! \\ Scanned image path
scanJobDidFinish	_ job: BRScanJob \\ Delegator result: BRScanJobResult \\ Completed job status as a BRScanJobResult enum

Example delegate methods are below:

```
extension ScanViewController: BRScanJobDelegate {
      func scanJob(_ job: BRScanJob!, progress: Float) {
            DispatchQueue.main.async { [weak self] in
                  self?.progressView.setProgress(progress, animated: true)
            }
      }

      // Note: This will display scanned images as they become available
      func scanJob(_ job: BRScanJob!, didFinishPage path: String!) {
            DispatchQueue.main.async { [weak self] in
                  if let scannedImage = UIImage(contentsOfFile: path) {
                        self?.scannedImageView.image = scannedImage
                  }
            }
      }

      func scanJobDidFinish(_ job: BRScanJob!, result: BRScanJobResult) {
            DispatchQueue.main.async { [weak self] in
                  self?.statusLabel.text = result == .success ? "OK" : “Nope”
            }
      }
}
```

A ScanJob instance contains several useful properties, described below:

Property	Type	Description
IPAddress	String!	The IP of the device
filePaths	[Any!]	An array of scanned image file paths as strings
options	[AnyHashable: Any!]	A dictionary of scan job options, defined below
error	NSError!	An NSError enum type, defined below

Job options are specified in a dictionary, much like the how scanner capabilities are reported in BRScanDevice.

Key	Value Type	Description
BRScanJobOptioncolorTypeKey	UInt (use the rawValue of a BRScanJobOptionColorType)	Configures the scanner with your color handling choice*
BRScanJobOptionDocumentSizeKey	UInt (use the rawValue of a BRScanJobOptionDocumentSize)	Specifies the document size of the scan**
BRScanJobOptionDuplexKey	UInt (use the rawValue of a BRScanJobOptionDuplex)	Configures the duplexing option for the job***
BRScanJobOptionSkipBlankPageKey	UInt (Use 0 for false, 1 for true)	Specifies if blank pages should generate a scanned image file

*enum **BRScanJobColorType**
- BRScanJobColorTypeColor
- BRScanJobColorTypeColorHighSpeed
- BRScanJobColorTypeGrayscale

enum **BRScanJobOptionDocumentSize
- BRScanJobOptionDocumentSizeAuto
- BRScanJobOptionDocumentSizeA3
- BRScanJobOptionDocumentSizeLedger
- BRScanJobOptionDocumentSizeJISB4
- BRScanJobOptionDocumentSizeLegal
- BRScanJobOptionDocumentSizeA4
- BRScanJobOptionDocumentSizeLetter
- BRScanJobOptionDocumentSizeJISB5
- BRScanJobOptionDocumentSizeBusinessCard
- BRScanJobOptionDocumentSizePhoto
- BRScanJobOptionDocumentSizePhotoL

***enum **BRScanJobOptionDuplex**
- BRScanJobOptionDuplexOff
- BRScanJobOptionDuplexLongEdge
- BRScanJobOptionDuplexShortEdge

The **error** property may contain a NSError enum detailing a problem with the scan job, otherwise nil is stored. The reported errors are:

- BRScanJobErrorConnection
- BRScanJobErrorMemory
- BRScanJobErrorDisk
- BRScanJobErrorPaperJam
- BRScanJobErrorTimeout
- BRScanJobErrorCoverOpen

- BRScanJobErrorDeviceBusy
- BRScanJobErrorDeviceMemory
- BRScanJobErrorNoPaper
- BRScanJobErrorTooLongDocument
- BRScanJobErrorFunctionLocked
- BRScanJobErrorADFCoverOpen
- BRScanJobErrorPapersOnTray
- BRScanJobErrorTooWideDocument
- BRScanJobErrorUnsupportedDocument
- BRScanJobErrorMultifeed
- BRScanJobErrorCardTrayOpen
- BRScanJobErrorCardTrayClose
- BRScanJobErrorADFUnsupportedSize
- BRScanJobErrorCardScanTopCoverOpen
- BRScanJobErrorCardScanTopCoverOpenDuringScan
- BRScanJobErrorCardScanSlotInsert
- BRScanJobErrorCardScanSlotEmpty
- BRScanJobErrorCardScanSlotJam
- BRScanJobErrorInternalParameter
- BRScanJobErrorInternalProcess
- BRScanJobErrorOther

An example job configuration is below:

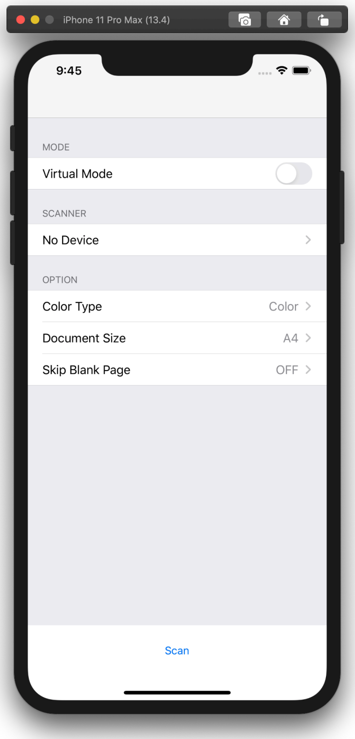
```
let scanJob = BRScanJob(ipAddress: selectedDevice.ipAddress)
var jobOptions = [String: UInt]()
jobOptions[BRScanJobOptionColorTypeKey] =
      BRScanJobOptionColorType.color.rawValue
jobOptions[BRScanJobOptionDocumentSizeKey] =
      BRScanJobOptionDocumentSize.letter.rawValue
jobOptions[BRScanJobOptionDuplexKey] =
      BRScanJobOptionDuplex.off.rawValue
jobOptions[BRScanJobOptionSkipBlankPageKey] = 0
scanJob?.options = jobOptions
```

Note: there are virtual job options as well, identical to the keys specified for the BRScanDeviceBrowser. Please refer to the browser documentation above for more details.

Workflow and Project Structure

Device Browsing

To get a sense of how to construct a scanning application with Swift, core concepts from **SampleApp**, which is included in the BRScanKit project and written in Objective-C, will be mimicked below. This is not a comprehensive tutorial on scanner app development—just a general and limited guide to constructing a Swift app with BRScanKit.



MODE
Virtual Mode

SCANNER
No Device

OPTION
Color Type Color
Document Size A4
Skip Blank Page OFF

In SampleApp, the BRSelectDeviceViewController class is where a user selects a scanner for configuration and use. To build a similar class with Swift, begin by importing the **BRScanKit**.

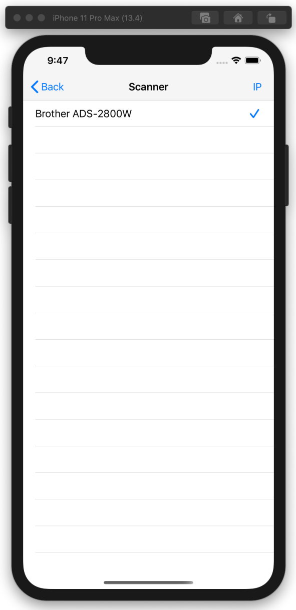
```swift
import BRScanKit
```

Add a property to hold an instance of **BRScanDeviceBrowser** and another property to hold a collection of **BRScanDevice**.

```swift
private let brotherBrowser = BRScanDeviceBrowser()
private var brotherDevices = [BRScanDevice]()
```

Assign the BRScanDeviceBrowser delegate to your Swift class. Usually in the UIViewController class viewDidLoad() method.

```swift
brotherBrowser.delegate = self
```



Brother ADS-2800W ✓

Start device searching with the **search()** method. Placing this in the viewWillAppear() method is convenient and useful if the browser will be used to populate a tableview. If you need to refresh the device collection, consider removing all devices before a subsequent search. This will prevent duplicate device listings.

```swift
brotherDevices.removeAll()
brotherBrowser.search()
```

To stop the search process, use the stop() method. This may be placed in the viewWillDisappear() method.

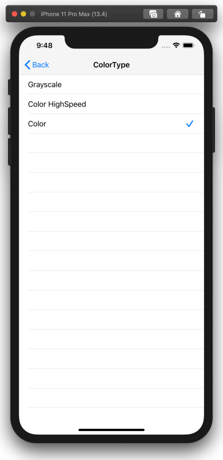
```swift
brotherBrowser.stop()
```

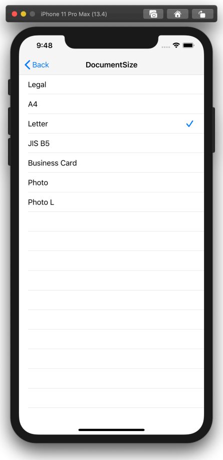
You will need to supply delegate methods to the browser. Here the delegates are placed in an extension for a view controller. The refresh() method is used to asynchronously update a tableview. You may notice that the scanDeviceBrowser(_:didRemove:) method is of limited use here; the device list is managed instead in the viewWillAppear() method for the view. Testing has revealed that it is uncertain when the delegated didRemove method is called by the browser.

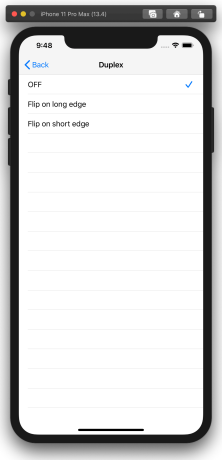
```
extension ViewController: BRScanDeviceBrowserDelegate {
      func scanDeviceBrowser(_ browser: BRScanDeviceBrowser!,
            didFind device: BRScanDevice!) {

            brotherDevices.append(device)
            refresh()
      }

      func scanDeviceBrowser(_ browser: BRScanDeviceBrowser!,
            didRemove device: BRScanDevice!) {

            guard let index = brotherDevices.firstIndex(of: device)
            else { return }

            brotherDevices.remove(at: index)
      }
}
```

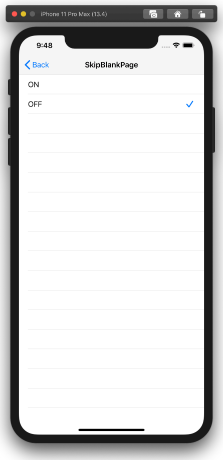
Recall that a BRScanDevice contains many useful properties that you can use in developing a detailed user interface. Also, the IP address will be needed when building a scanning job.

Configuring a Scanning Job









Device configuration in SampleApp is found in the BRScanViewController class. This view contains parent/child tableviews for option settings. How you choose to set **BRScanJob** options is entirely in your control, but you will need to accomplish two things always: supply a valid IP address for a scanner and set dictionary options.

As before, you will need access to the framework.

```
import BRScanKit
```

Create an instance of a BRScanJob and complete option settings. In this example the settings are hard coded and unavailable to user control. Please keep in mind that you may have to limit user control options based on the characteristics of the scanner in use. Examine the properties returned from BRScanDevice—it can be very useful in limiting options to your user and avoiding scanning errors due to unavailable features.

```
let scanJob = BRScanJob(ipAddress: selectedDevice.ipAddress)

var jobOptions = [String: UInt]()
jobOptions[BRScanJobOptionColorTypeKey] =
      BRScanJobOptionColorType.color.rawValue
jobOptions[BRScanJobOptionDocumentSizeKey] =
      BRScanJobOptionDocumentSize.letter.rawValue
jobOptions[BRScanJobOptionDuplexKey] = BRScanJobOptionDuplex.off.rawValue
jobOptions[BRScanJobOptionSkipBlankPageKey] = 0

scanJob?.options = jobOptions
```

Running a Job

In SampleApp, scanning occurs in the BRScanResultViewController class. Scanned images are displayed in a UICollectionView, which requires that images are indexed inside of an array. To achieve the same in Swift, begin by importing the framework.

```
import BRScanKit
```

Add a property to an instance of **BRScanJob** and build an array to hold paths to the scanned images. The scanJob property will need all of the job configuration that was built in another view. Be sure to pass those forward to avoid errors. This example skips those steps.

```
var scanJob = BRScanJob()
var scannedImages = [UIImage]()
```

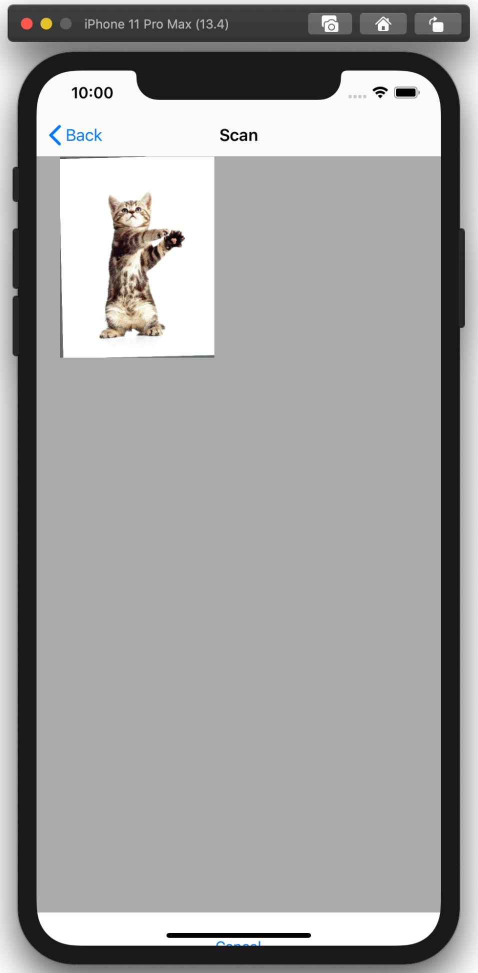
Assign a delegate to the view class.

```
scanJob.delegate = self
```

With the scan job configured, use the **start()** and **cancel()** methods to control device scanning.

```
scanJob.start()
```

If your scanner has a document staged, it should immediately begin scanning. You should provide an option to cancel the scan in your user interface. The scanner will report errors, but your user may notice problems before the error is triggered.



Delegate methods are required to handle the output of the scanning operation. SampleApp does not update a progress indicator but an example is included here for completeness. Notice that all updates are handled asynchronously so that UI controls can refresh views as the scanner processes.

The **filePaths** property will contain an array of file paths for each scanned image at the end of the job run. Internally, this is stored as an Any type, so you will need to cast it to a string or URL in order to use it.

Special note: errors do not throw exceptions. You must examine the **result** parameter in **scanDidFinish** to know final job status.* The **job.error** property will provide more details about an error.

```
extension ScanViewController: BRScanJobDelegate {
      // Periodically update progress during the scanning process.
      func scanJob(_ job: BRScanJob!, progress: Float) {
            DispatchQueue.main.async { [weak self] in
                  self?.progressView.setProgress(progress, animated: true)
            }
      }

      // Process your scanned image with this method.
      func scanJob(_ job: BRScanJob!, didFinishPage path: String!) {
            DispatchQueue.main.async { [weak self] in
                  if let scannedImage = UIImage(contentsOfFile: path) {
                        self?.scannedImages.append(scannedImage)
                  }
            }
      }

      // Report the result of the scanning job and handle end of job
      // activities here.
      // Notice that this is non-throwing. Examine the BRScanJobResult enum
      // for status. Look inside job.error for more details.
      func scanJobDidFinish(_ job: BRScanJob!, result: BRScanJobResult) {
            DispatchQueue.main.async { [weak self] in
                  self?.progressView.isHidden = true
                  // TODO: Examine result
                  // TODO: Examine job.error
            }
      }
}
```

*enum **BRScanJob**
- BRScanJobResultSuccess
- BRScanJobResultCancel
- BRScanJobResultFail