
Camera



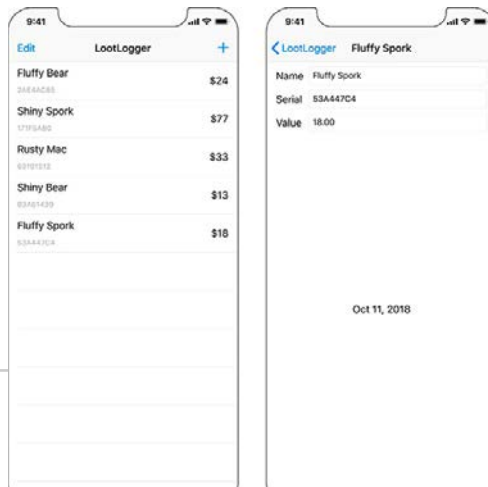
Camera

Adding a camera to existing LootLogger app.

`UIImagePickerController` enables the user to take and save a picture of each item. The image will then be associated with an Item instance and viewable in the item's detail view.

`ImageStore`, a second store for images, will store images (large data), fetch and cache images as they are needed.

the `DetailViewController` get and display an image. An easy way to display an image is to put an instance of `UIImageView` on the screen.

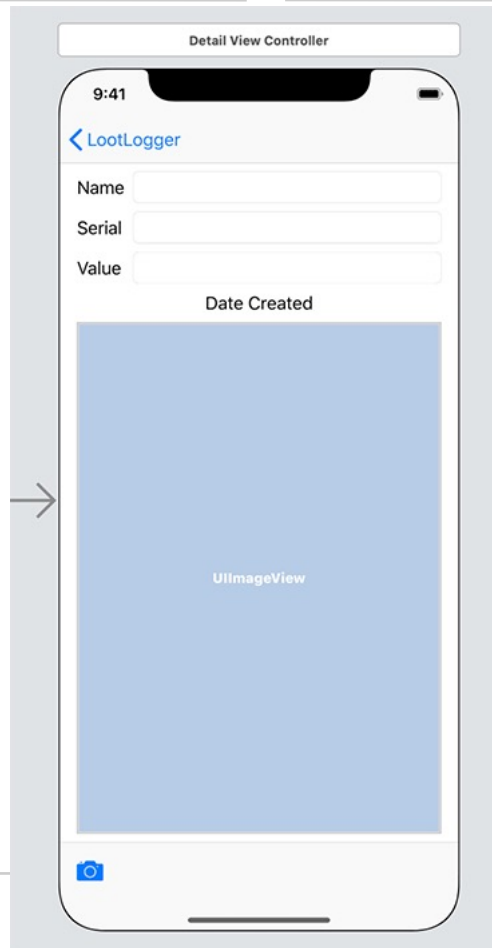


Camera

Open LootLogger.xcodeproj and Main.storyboard. Drag an Image View from the library onto the detail view controller's view, positioning it as the last view within the stack view.

Select the image view and open its size inspector. You want the vertical content hugging and content compression resistance priorities for the image view to be lower than those of the other views.

- Change the Vertical Content Hugging Priority to 248
- Vertical Content Compression Resistance Priority to 749.



Camera

A `UIImageView` displays an image according to the image view's `contentMode` property. This property determines where to position and how to resize the content within the image view's frame.

For image views, you will usually want either **aspect fit** (if you want to see the whole image) or **aspect fill** (if you want the image to fill the image view).

With the `UIImageView` still selected, open the attributes inspector. Find the Content Mode attribute and confirm it is set to Aspect Fit.



Aspect Fill



Aspect Fit

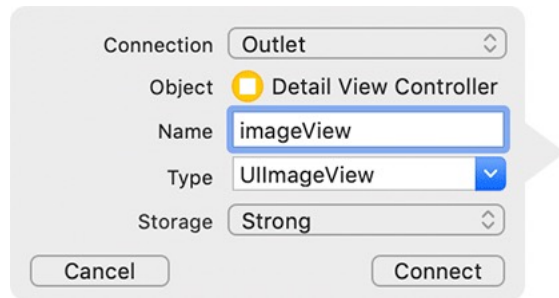
Camera

Option-click DetailViewController.swift in the project navigator to open it in another editor. Control-drag from the UIImageView to the top of DetailViewController.swift. Name the outlet imageView and make sure the storage type is Strong. Click Connect.

...

```
class DetailViewController: UIViewController, UITextFieldDelegate {
```

```
    @IBOutlet var nameField: UITextField!  
    @IBOutlet var serialNumberField: UITextField!  
    @IBOutlet var valueField: UITextField!  
    @IBOutlet var dateLabel: UILabel!  
    @IBOutlet var imageView: UIImageView!
```



Camera

In the `choosePhotoSource(_:)` method, you will instantiate a `UIImagePickerController` and present it on the screen. When creating an instance of `UIImagePickerController`, you must set its `sourceType` property and assign it a delegate.

Close the editor showing `Main.storyboard`. In `DetailViewController.swift`, add a new method that creates and configures a `UIImagePickerController` instance. You will need to create the `UIImagePickerController` instance from more than one place, so abstracting it into a method will help avoid repetition.

```
func imagePicker(for sourceType: UIImagePickerController.SourceType)
    -> UIImagePickerController {

    let imagePicker = UIImagePickerController()
    imagePicker.sourceType = sourceType
    return imagePicker
}
```

Camera

“The sourceType is a UIImagePickerController.

SourceType enumeration value and tells the image picker where to get images. It has three possible values:

`.camera`

Allows the user to take a new photo.

`.photoLibrary`

Prompts the user to select an album and then a photo from that album.

`.savedPhotosAlbum`

Prompts the user to choose from the most recently taken photos.



Camera

In `choosePhotoSource(_)`, create an image picker controller instance when the user chooses one of the action sheet options.

```
@IBAction func choosePhotoSource(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil,
                                         message: nil,
                                         preferredStyle: .actionSheet)

    alertController.modalPresentationStyle = .popover
    alertController.popoverPresentationController?.barButtonItem = sender
    let cameraAction = UIAlertAction(title: "Camera", style: .default) { _ in
        print("Present camera")
        let imagePicker = self.imagePicker(for: .camera)
    }
    alertController.addAction(cameraAction)

    let photoLibraryAction =
        UIAlertAction(title: "Photo Library", style: .default) { _ in
            print("Present photo library")
            let imagePicker = self.imagePicker(for: .photoLibrary)
        }
    alertController.addAction(photoLibraryAction)
    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
    alertController.addAction(cancelAction)
    present(alertController, animated: true, completion: nil)
}
```



Camera

The first source type, `.camera`, will not work on a device that does not have a camera. So before using this type, you have to check for a camera by calling the method `isSourceTypeAvailable(_:)` on the `UIImagePickerController` class:

```
class func isSourceTypeAvailable  
(
```

Calling this method returns a Boolean value indicating whether the device supports the passed-in source type.

Update `choosePhotoSource(_:)` to only show the camera option if the device has a camera:

Camera

Update choosePhotoSource():

```
@IBAction func choosePhotoSource(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil,
                                         message: nil,
                                         preferredStyle: .actionSheet)

    alertController.modalPresentationStyle = .popover
    alertController.popoverPresentationController?.barButtonItem = sender

    if UIImagePickerController.isSourceTypeAvailable(.camera) {
        let cameraAction = UIAlertAction(title: "Camera", style: .default) { _ in
            let imagePicker = self.imagePicker(for: .camera)
        }
        alertController.addAction(cameraAction)
    }
}
```

Camera

In addition to a source type, the `UIImagePickerController` instance needs a delegate. When the user selects an image from the `UIImagePickerController`'s interface, the delegate is sent the message `imagePickerController(_:didFinishPickingMediaWithInfo:)`. (If the user taps the cancel button, then the delegate receives the message `imagePickerControllerDidCancel(_:)`.) The image picker's delegate will be the instance of `DetailViewController`.

In `DetailViewController.swift`:

```
class DetailViewController: UIViewController, UITextFieldDelegate,
    UINavigationControllerDelegate, UIImagePickerControllerDelegate {

func imagePicker(for sourceType: UIImagePickerController.SourceType) ->
    UIImagePickerController {
    let imagePicker = UIImagePickerController()
    imagePicker.sourceType = sourceType
    imagePicker.delegate = self
    return imagePicker
}
```

Camera

In choosePhotoSource(_:) add the following code:

```
if UIImagePickerController.isSourceTypeAvailable(.camera) {
    let cameraAction = UIAlertAction(title: "Camera", style: .default) { _ in
        let imagePicker = self.imagePicker(for: .camera)
        self.present(imagePicker, animated: true, completion: nil)
    }
    alertController.addAction(cameraAction)
}
```

```
let photoLibraryAction = UIAlertAction(title: "Photo Library", style: .default) { _ in
    let imagePicker = self.imagePicker(for: .photoLibrary)
    self.present(imagePicker, animated: true, completion: nil)
}
alertController.addAction(photoLibraryAction)
```

Camera

Apple's documentation for UIImagePickerController mentions that the camera should be presented full screen, and the photo library and saved photos album must be presented in a popover. The only change you need to make to satisfy these requirements is to present the photo library in a popover.

Update the image picker to do just that.

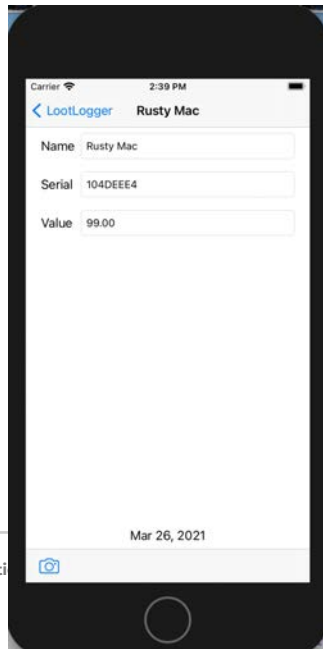
In `DetailViewController.swift`:

```
let photoLibraryAction = UIAlertAction(title: "Photo Library", style: .default) { _ in
    let imagePicker = self.imagePicker(for: .photoLibrary)
    imagePicker.modalPresentationStyle = .popover
    imagePicker.popoverPresentationController?.barButtonItem = sender
    self.present(imagePicker, animated: true, completion: nil)
}
```

Camera

Build and run the application. Select an Item to see its details and then tap the camera button on the UIToolbar. Choose Photo Library and then select a photo.

Camera option no longer appears, because the simulator has no camera.



Camera

When attempting to access potentially private information, such as the camera, iOS prompts the user to consent to that access. Contained within this prompt is a description of why the application wants to access the information. LootLogger is missing this description, and therefore the application is crashing.

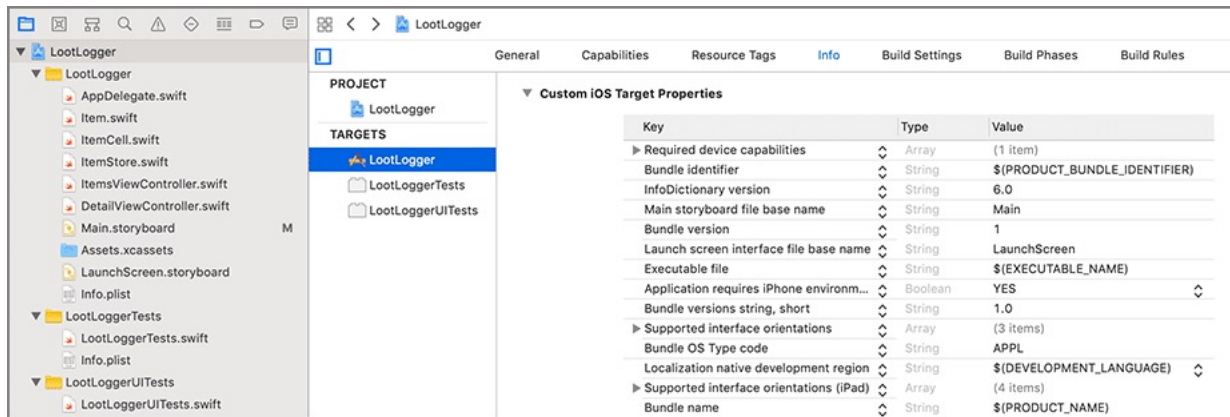
There are a number of capabilities on iOS that require user approval before use:

- Camera
- Photos
- Location
- microphone
- HealthKit data
- Calendar
- Reminders

For each of these, your application must supply a usage description that specifies the reason that your application wants to access the capability or information. This description will be presented to the user when the application attempts the access.

Camera

In the project navigator, select the project at the top. In the editor, make sure the LootLogger target is selected and open the Info tab along the top

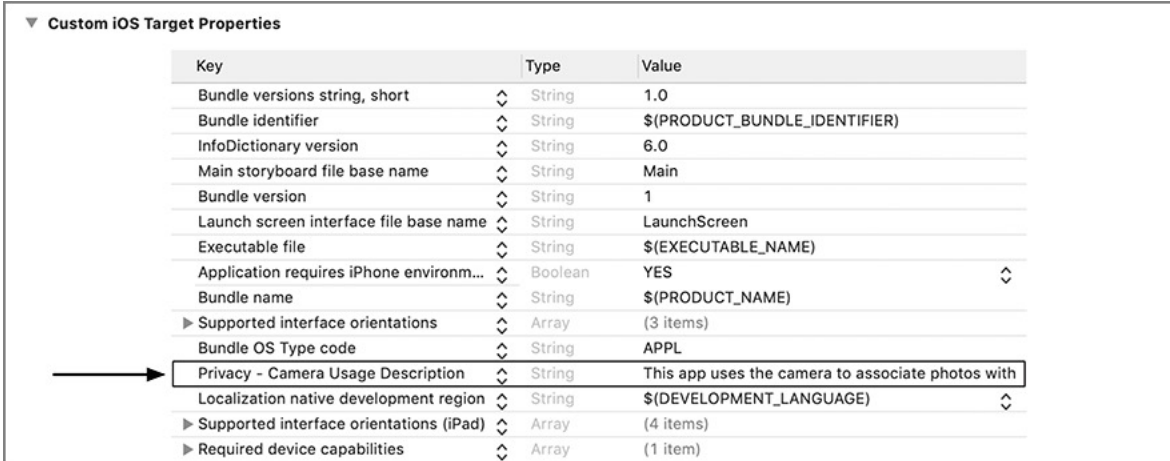


Hover over the last entry in this list of Custom iOS Target Properties and click the + button. Set the Key of the new entry that appears to **NSCameraUsageDescription** and the Type to **String**. When you press Return, the key name in Xcode will change from **“NSCameraUsageDescription”** to **“Privacy – Camera Usage Description.”** By default, Xcode displays human-readable strings instead of the actual key names. When adding or editing an entry, you can use either the human-readable string or the actual key name.

Camera

Control-click on the key-like to view the actual key names in Xcode, Control-click on the key-value table and select Raw Keys & Values.

For the Value, enter This app uses the camera to associate photos with items. This is the string that will be presented to the user.



▼ Custom iOS Target Properties

Key	Type	Value
Bundle versions string, short	String	1.0
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Main storyboard file base name	String	Main
Bundle version	String	1
Launch screen interface file base name	String	LaunchScreen
Executable file	String	\$(EXECUTABLE_NAME)
Application requires iPhone environ...	Boolean	YES
Bundle name	String	\$(PRODUCT_NAME)
▶ Supported interface orientations	Array	(3 items)
Bundle OS Type code	String	APPL
Privacy - Camera Usage Description	String	This app uses the camera to associate photos with
Localization native development region	String	\$(DEVELOPMENT_LANGUAGE)
▶ Supported interface orientations (iPad)	Array	(4 items)
▶ Required device capabilities	Array	(1 item)

Camera

Build and run the application **on a device** and navigate to an item. Tap the camera button, select the Camera option, and you will see the permission dialog presented with the usage description that you provided. After you tap OK, the UIImagePickerController's camera interface will appear on the screen, and you can take a picture.



Saving the Image

Selecting an image dismisses the UIImagePickerController and returns you to the detail view. However, you do not have a reference to the photo once the image picker is dismissed. To fix this, you are going to implement the delegate method `imagePickerController(_:didFinishPickingMediaWithInfo:)`. This method is called on the image picker's delegate when a photo has been selected.

In `DetailViewController.swift`, implement `imagePickerController(_:didFinishPickingMediaWithInfo:)` to put the image

Saving the Image

Create a new Swift file named ImageStore. In ImageStore.swift, define the ImageStore class and add a property t

```
import UIKit
```

```
class ImageStore {
```

```
    let cache = NSCache<NSString,UIImage>()
```

```
}
```



Saving the Image

The cache works very much like a dictionary. You are able to add, remove, and update values associated with a given key. Unlike a dictionary, the cache will automatically remove objects if the system gets low on memory.

Note that the cache is associating an instance of `NSString` with `UIImage`. `NSString` is Objective-C's version of `String`. Due to the way `NSCache` is implemented (it is an Objective-C class, like most of Apple's classes that you have been working with), it requires you to use `NSString` instead of `String`.

Saving the Image

In the `ImageStore.swift`:

```
class ImageStore {  
  
    let cache = NSCache<NSString, UIImage>()  
  
    func setImage(_ image: UIImage, forKey key: String) {  
        cache.setObject(image, forKey: key as NSString)  
    }  
  
    func image(forKey key: String) -> UIImage? {  
        return cache.object(forKey: key as NSString)  
    }  
  
    func deleteImage(forKey key: String) {  
        cache.removeObject(forKey: key as NSString)  
    }  
  
}
```



Saving the Image

In `DetailViewController.swift`, add a property for an `ImageStore`:

```
var item: Item! {  
    didSet {  
        navigationItem.title = item.name  
    }  
}  
var imageStore: ImageStore!
```

in `ItemsViewController.swift`:

```
var itemStore: ItemStore!  
var imageStore: ImageStore!
```

Saving the Image

in `ItemsViewController.swift`:

update `prepare(for:sender:)` to set the `imageStore` property on `DetailViewController`:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // If the triggered segue is the "showItem" segue
    switch segue.identifier {
    case "showItem":
        // Figure out which row was just tapped
        if let row = tableView.indexPathForSelectedRow?.row {

            // Get the item associated with this row and pass it along
            let item = itemStore.allItems[row]
            let detailViewController
                = segue.destination as! DetailViewController
            detailViewController.item = item
            detailViewController.imageStore = imageStore
        }
    default:
        preconditionFailure("Unexpected segue identifier.")
    }
}
```


Saving the Image

In SceneDelegate.sw

```
func scene(_ scene: UIScene,  
    willConnectTo session: UISceneSession,  
    options connectionOptions: UIScene.ConnectionOptions) {  
    guard let _ = (scene as? UIWindowScene) else { return }
```

```
// Create an ImageStore  
let imageStore = ImageStore()
```

```
// Create an ItemStore  
let itemStore = ItemStore()
```

Saving the Image

When an image is added to the store, it will be put into the cache under a unique key, and the associated Item object will be given that key. When the DetailViewController wants an image from the store, it will ask its item for the key and search the cache for the image.

Add a property to Item.swift to store the key.

```
let dateCreated: Date
```

```
let itemKey: String
```

Saving the Image

In `Item.swift`, generate a UUID and set it as the `itemKey`.

```
init(name: String, serialNumber: String?, valueInDollars: Int) {  
    self.name = name  
    self.valueInDollars = valueInDollars  
    self.serialNumber = serialNumber  
    self.dateCreated = Date()  
    self.itemKey = UUID().uuidString  
}
```

Saving the Image

in [DetailViewController.swift](#), update `imagePickerController(_:didFinishPickingMediaWithInfo:)` to store the image in the `ImageStore`:

```
func imagePickerController(_ picker:
    UIImagePickerController,
    didFinishPickingMediaWithInfo info:
    [UIImagePickerController.InfoKey: Any]) {
    // Get picked image from info dictionary
    let image = info[UIImagePickerControllerOriginalImage] as! UIImage

    // Store the image in the ImageStore for the item's key
    imageStore.setImage(image, forKey: item.itemKey)

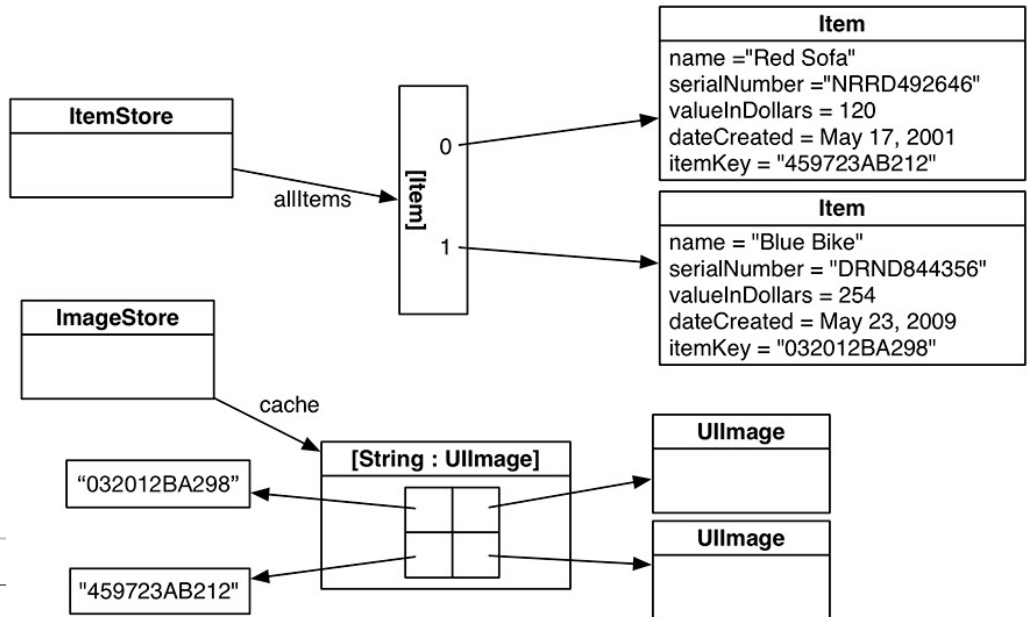
    // Put that image on the screen in the image view
    imageView.image = image

    // Take image picker off the screen - you must call this dismiss method
    dismiss(animated: true, completion: nil) }
```

Saving the Image

Each time an image is captured, it will be added to the store. Notice that the images are saved immediately after being taken, while the instances of Item are saved only when the application enters the background. You save the images right away because they are too big to keep in memory for long.

Both the ImageStore and the Item will know the key for the image, so both will be able to access it as needed.



Saving the Image

In `ItemsViewController.swift`, update `tableView(_:commit:forRowAt:)` to remove the item's image from the image store:

```
override func tableView(_ tableView: UITableView,
                        commit editingStyle: UITableViewCellEditingStyle,
                        forRowAt indexPath: IndexPath) {
    // If the table view is asking to commit a delete command...
    if editingStyle == .delete {
        let item = itemStore.allItems[indexPath.row]

        // Remove the item from the store
        itemStore.removeItem(item)

        // Remove the item's image from the image store
        imageStore.deleteImage(forKey: item.itemKey)

        // Also remove that row from the table view with an animation
        tableView.deleteRows(at: [indexPath], with: .automatic)
    }
}
```

Saving the Image

Each item's `itemKey` is encoded and decoded, but what about its image? At the moment, images are lost when the app enters the background state. In this section, you will extend the image store to save images as they are added and fetch them as they are needed.

The images for `Item` instances should also be stored in the Documents directory. You can use the image key generated when the user takes a picture to name the image in the filesystem.

Implement a new method in `ImageStore.swift` named `imageURL(forKey:)` to create a URL in the documents directory using a given key.

Saving the Image

In ImageStore.swift:

```
func imageURL(forKey key: String) -> URL {  
    let documentsDirectories =  
        FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)  
    let documentDirectory = documentsDirectories.first!  
    return documentDirectory.appendingPathComponent(key)  
}
```


Saving the Image

In `ImageStore.swift`:

modify `setImage(_:forKey:)` to get a URL and save the image:

```
func setImage(_ image: UIImage, forKey key: String) {
    cache.setObject(image, forKey: key as NSString)

    // Create full URL for image
    let url = imageURL(forKey: key)

    // Turn image into JPEG data
    if let data = image.jpegData(compressionQuality: 0.5) {
        // Write it to full URL
        try? data.write(to: url)
    }
}
```

Saving the Image

In `ImageStore.swift`:

update `image(forKey:)` so that the `ImageStore` will load the image from the filesystem if it does not already have it:

```
func image(forKey key: String) -> UIImage? {  
—return cache.object(forKey: key as NSString)  
  
    if let existingImage = cache.object(forKey: key as NSString) {  
        return existingImage  
    }  
  
    let url = imageURL(forKey: key)  
    guard let imageFromDisk = UIImage(contentsOfFile: url.path) else {  
        return nil  
    }  
  
    cache.setObject(imageFromDisk, forKey: key as NSString)  
    return imageFromDisk  
}
```

Saving the Image

You are able to save an image to disk and retrieve an image from disk. Now you need the functionality to remove an image from disk. In ImageStore.swift, make sure that when an image is delete

```
func deleteImage(forKey key: String) {
    cache.removeObject(forKey: key as NSString)

    let url = imageURL(forKey: key)
    do {
        try FileManager.default.removeItem(at: url)
    } catch {
        print("Error removing the image from disk: \(error)")
    }
}
```

Saving the Image

Now that the ImageStore can store images, and instances of Item have a key to get an image, you need to teach DetailViewController how to grab the image for the selected Item and place it in its imageView.

The DetailViewController's view will appear when the user taps a row in ItemsViewController and when the UIImagePickerController is dismissed. In both of these situations, the imageView should be populated with the image of the Item being displayed. Currently, it is only happening when the UIImagePickerController is dismissed.

Saving the Image

In `DetailViewController.swift`,

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    nameField.text = item.name
    serialNumberField.text = item.serialNumber
    valueField.text =
numberFormatter.string(from: NSNumber(value: item.valueInDollars))
    dateLabel.text = dateFormatter.string(from: item.dateCreated)

    // Get the item key
    let key = item.itemKey

    // If there is an associated image with the item, display it on the image view
    let imageToDisplay = imageStore.image(forKey: key)
    imageView.image = imageToDisplay
}
```

Saving the Image

Build and run the application. Create an item and select it from the table view. Then tap the camera button and select a picture. The image will appear as it should. Pop out from the item's details to the list of items. Unlike before, if you tap and drill down to see the details of the item you added a picture to, you will see the image.