Scene States

Saving, Loading
Presenting View Controllers

Daria Tsoupikova
Sabine Krauss

Saving and Loading Data

Several ways to save and load data in an iOS application.
Most common mechanisms for writing to or reading from the filesystem in iOS.

MVC model:

- Model objects are responsible for holding on to the data that the user manipulates.
- View objects reflect that data
- Controllers are responsible for keeping the views and the model objects in sync.

Saving and loading “data” almost always means saving and loading model objects.

In LootLogger, the model objects that a user manipulates are instances of Item. For LootLogger to be a useful application, instances of Item must persist between runs of the application. We can make the Item type codable so that instances can be saved to and loaded from disk.

Declaring conformance to Codable

Codable types conform to the Encodable and Decodable protocols and implement their required methods, which are encode(to:) and init(from:)

Open LootLogger and add this conformance in Item.swift.

```
class Item: Equatable, Codable {
```

Declaring conformance to Codable

Now that Item can be encoded and decoded, you will need a coder. A coder is responsible for encoding a type into some external representation. Built-in coder: PropertyListEncoder saves data out in a property list format. A property list is a representation of data that can be saved to disk and read back in at a later point.

In ItemStore.swift, implement a new method that will be responsible for saving the items.

```
        func saveChanges() -> Bool {

//an instance of PropertyListEncoder
let encoder = PropertyListEncoder()
/*call the encode(_:) method on that encoder, passing the allItems array, which will encode each of the Item*/
let data = encoder.encode(allItems)

    return false
}
```

Declaring conformance to Codable

The Swift compiler is telling you that you are not handling a possible error when attempting to encode allItems.

In ItemStore.swift, update saveChanges() to call encode(_:) using a do-catch statement:

```
func saveChanges() -> Bool {
    do {
        let encoder = PropertyListEncoder()
        let data = encoder.encode(allItems)
        let data = try encoder.encode(allItems)
    } catch {

    }

    return false
}
```

If a method does throw an error, then the program immediately exits the do block; no further code in the do block is executed.

Declaring conformance to Codable

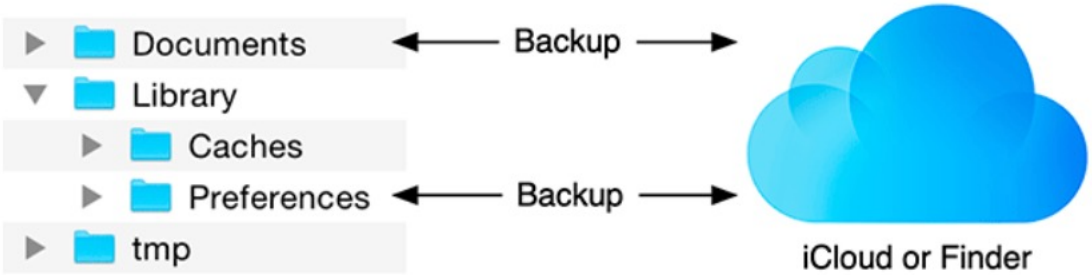
Next, update saveChanges() to print out the error to the console and to use an explicit name for the error being caught:

```
    func saveChanges() -> Bool {


  do {
      let encoder = PropertyListEncoder()
      let data = try encoder.encode(allItems)
  } catch let encodingError {
    print("Error encoding allItems: \(encodingError)")
  }


  return false
}
```

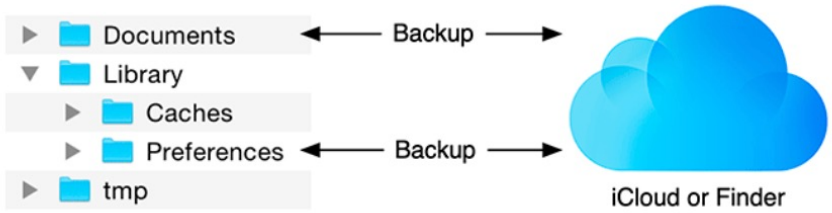
App Sandbox

Now you need to persist this data to disk. You can build the application now to make sure there are no syntax errors, but you do not yet have a way to kick off the saving and loading. You also need a place on the filesystem to store the saved items.

Every iOS application has its own application sandbox. An application sandbox is a directory on the filesystem that is barricaded from the rest of the filesystem. Your application must stay in its sandbox, and no other application can access its sandbox.



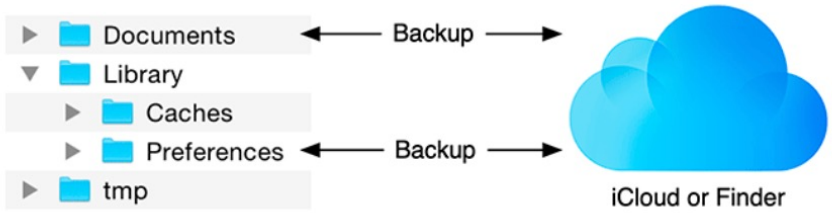
App Sandbox



Documents/
This directory is where you write data that the application generates during runtime and that you want to persist between runs of the application. It is backed up when the device is synchronized with iCloud or Finder. If something goes wrong with the device, files in this directory can be restored from iCloud or Finder. In LootLogger, the file that holds the data for all your items will be stored here.

Library/Caches/
This directory is where you write data that the application generates during runtime and that you want to persist between runs of the application. However, unlike the Documents directory, it does not get backed up when the device is synchronized with iCloud or Finder.

App Sandbox



Library/Preferences/
This directory is where any preferences are stored and where the Settings application looks for application preferences. Library/Preferences is handled automatically by the class UserDefaults and is backed up when the device is synchronized with iCloud or Finder.

tmp/
This directory is where you write data that you will use temporarily during an application's runtime. The OS may purge files in this directory when your application is not running. However, to be tidy you should explicitly remove files from this directory when you no longer need them. This directory does not get backed up when the device is synchronized with iCloud or Finder.

App Sandbox

The instances of Item from LootLogger will be saved to a single file in the Documents/ directory. The ItemStore will handle writing to and reading from that file. To do this, the ItemStore needs to construct a URL to this file. You will have a place to save data on the filesystem and a model object that can be saved to the filesystem.

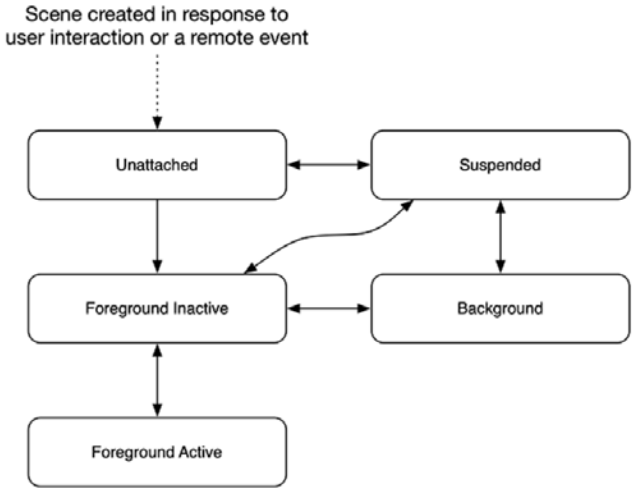
Implement a new property in **ItemStore.swift** to store this URL:

```
var allItems = [Item]()
let itemArchiveURL: URL = {
    let documentsDirectories =
        FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)
    let documentDirectory = documentsDirectories.first!
    return documentDirectory.appendingPathComponent("items.plist")
}()
```

Scene States and Transitions

On iPhone, there is only ever one scene – one instance of your application's UI. On iPad, users may have multiple scenes, and they might be visible simultaneously. Scenes can be created and destroyed as a user opens and closes windows, so you should think about the lifecycle of a scene in addition to the lifecycle of the application as a whole. For example, one scene can go into the background while another scene remains in the foreground.

In LootLogger, the items will be archived when the scene enters the background state. It is useful to understand the states a scene can be in as well as what causes a scene to transition between states and how your code can be notified of these transitions.



Scene States and Transitions

When a scene is not running, it is in the **unattached** state, and it does not execute any code or have any memory reserved in RAM.

After a scene is launched, it briefly enters the **foreground** inactive state before entering the foreground active state. When in the foreground active state, a scene's interface is on the screen, it is accepting events, and its code is handling those events.

While in the **active** state, a scene can be temporarily interrupted by a system event, like a phone call, or interrupted by a user event, like triggering Siri or opening the task switcher. At this point, the scene reenters the foreground inactive state. In the inactive state, a scene is usually visible and is executing code, but it is not receiving events. Scenes typically spend very little time in the inactive state.

When the user returns to the Home screen or switches to another application, the scene enters the **background** state. (Actually, it spends a brief moment in the foreground inactive state before transitioning to the background state.) In the background state, a scene's interface is not visible or receiving events, but it can still execute code. By default, a scene that enters the background state has about 10 seconds before it enters the **suspended** state.

Scene States and Transitions

To write data to the filesystem, you call the method write(to:options:) on an instance of Data. The first parameter indicates a location on the filesystem to write the data into, and the second parameter allows you to specify options that customize the writing behavior.

The item data will be persisted to disk when the saveChanges() method is called.

Scene States and Transitions

In ItemStore.swift, update saveChanges() to write out the data to the itemArchiveURL:

```
func saveChanges() -> Bool {
    print("Saving items to: \(itemArchiveURL)")

    do {
        let encoder = PropertyListEncoder()
        let data = try encoder.encode(allItems)
        try data.write(to: itemArchiveURL, options: [.atomic])
        print("Saved all of the items")
        return true
    } catch let encodingError {
        print("Error encoding allItems: \(encodingError)")
        return false
    }

    return false
}
```

Scene States and Transitions

For LootLogger, you will save the encoded data for instances of Item when the application "exits." When the user leaves the application (such as by going to the Home screen), the notification UIScene.didEnterBackgroundNotification is posted to the NotificationCenter. You will listen for that notification and save the items when it is posted.

It is important to understand that Notifications and the NotificationCenter are not associated with visual "notifications," like push and local notifications that the user sees when an alarm goes off or a text message is received.

Scene States and Transitions

In ItemStore.swift, override init() to add an observer for the UIScene.didEnterBackgroundNotification notification:

```
    init() {
  let notificationCenter = NotificationCenter.default
  notificationCenter.addObserver(self,
                                 selector:
#selector(saveChanges),
                                 name: UIScene.didEnterBackgroundNotification,
                                 object: nil)
}
```

Add @objc annotation to saveChanges() funciotn to handle Objective-C:

```
@objc func saveChanges() -> Bool {
```

Instead of `func saveChanges() -> Bool {`

Scene States and Transitions

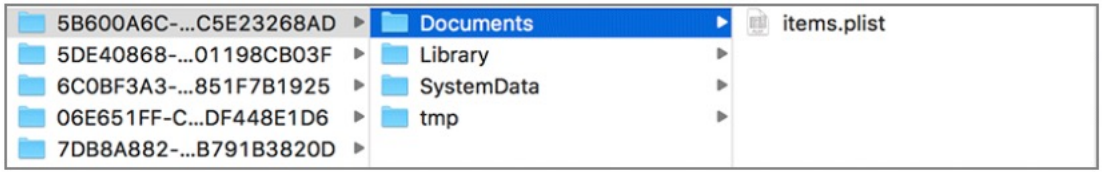
Build and run the application. Create a few instances of Item, then go to the simulator's Home screen, either by selecting Hardware → Home or with the keyboard shortcut Command-Shift-H. Check the Xcode console, and you should see a log statement indicating that the items were saved. (You may see additional log statements generated by iOS, which you can ignore.)

While you cannot yet load these instances of Item back into the application, you can still verify that something was saved.

In the console's log statements, find one that logs out the itemArchiveURL location and another that indicates whether saving was successful.

Scene States and Transitions

Open Finder and press Command-Shift-G. Paste the file path that you copied from the console, replacing the file:/// with just /, and press Return. You will be taken to the directory that contains the items.plist file. Press Command-Up to navigate to the parent directory of items.plist. This is the application's sandbox directory. Here, you can see the Documents/, Library/, and tmp/ directories alongside the application itself



Next : To load instances of Item when the application launches, you will use the PropertyListDecoder type when the ItemStore is created.

Scene States and Transitions

In ItemStore.swift, update init() to load in the items:

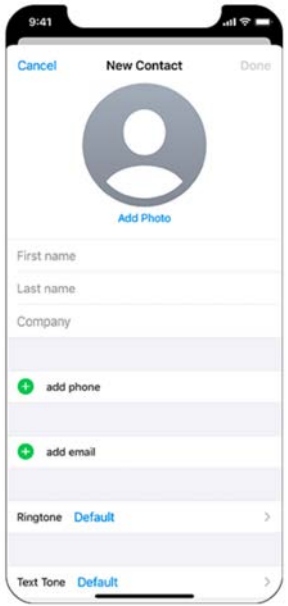
```
init() {
    do {
        let data = try Data(contentsOf: itemArchiveURL)
        let unarchiver = PropertyListDecoder()
        let items = try unarchiver.decode([Item].self, from: data)
        allItems = items
    } catch {
        print("Error reading in saved items: \(error)")
    }

    let notificationCenter = NotificationCenter.default
    notificationCenter.addObserver(self,
                                   selector: #selector(saveChanges),
                                   name: UIScene.didEnterBackgroundNotification,
                                   object: nil)
}
```

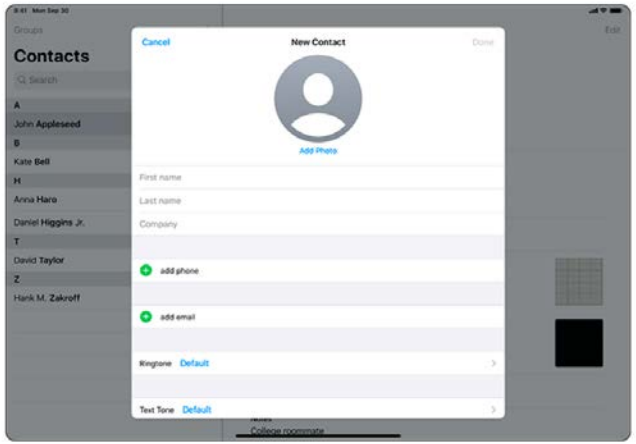
Scene States and Transitions

Build and run the application. Your items will be available until you explicitly delete them.

Presenting View Controllers

iOS apps often present users with a view controller showing an action they must complete or dismiss. For example, when adding a new contact on iPhone, users are presented with a screen to fill out the contact's details. We call this kind of presentation modal, as the application is being put into a different "mode" where a set of actions become the focus. The user must interact with the modally presented view controller before proceeding.



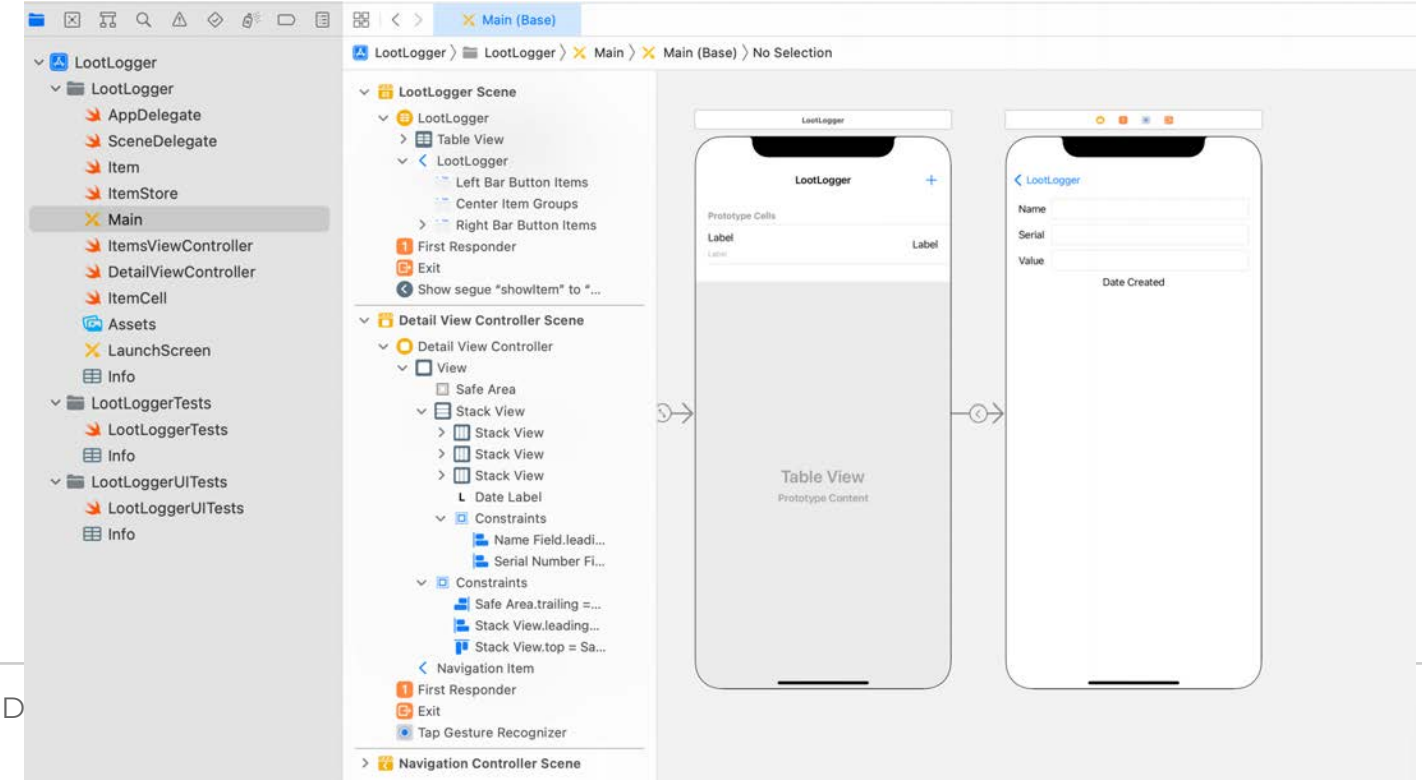


Presenting View Controllers

We will extend the LootLogger application to add the ability for users to associate a photo with each of their items. We will present the user with the option to select a photo from either the camera or the device's photo library.

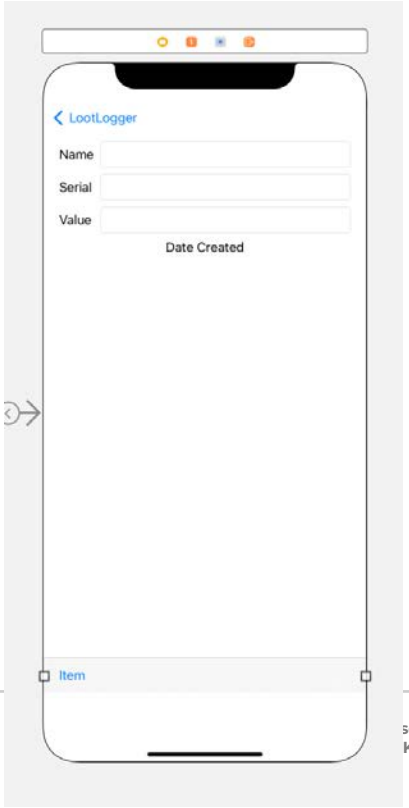
Presenting View Controllers

Open LootLogger.xcodeproj and navigate to Main.storyboard. In the detail view controller, select the bottom constraint for the outer stack view and press Delete to remove it. The stack view will resize itself, which will make some room for the toolbar at the bottom of the screen.



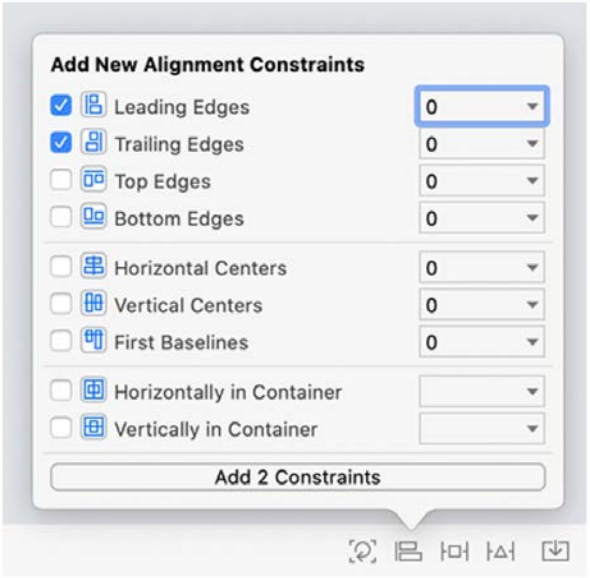
Presenting View Controllers

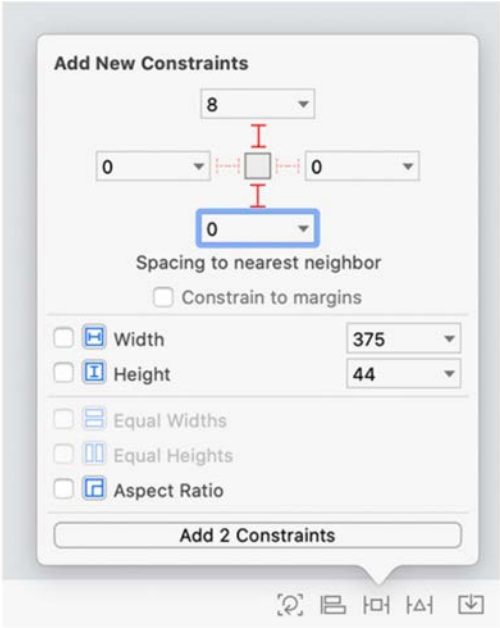
Now, drag a toolbar from the library and place it near the bottom of the view.
Make sure it is above the Home indicator (the black bar along the bottom of the screen).



Presenting View Controllers

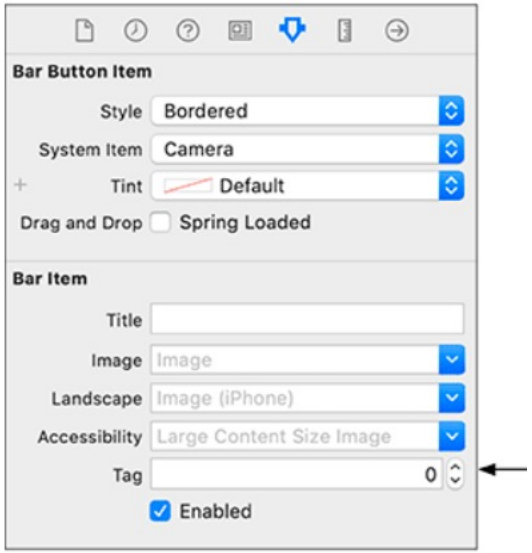
You want the toolbar to extend from the superview's leading edge to its trailing edge, independent of the safe area. To do this, select both the toolbar and the superview and open the Auto Layout Align menu. Configure the constraints as shown on the left:
Select only the toolbar this time and open the Auto Layout Add New Constraints menu. Configure the top and bottom constraints as shown on the right:

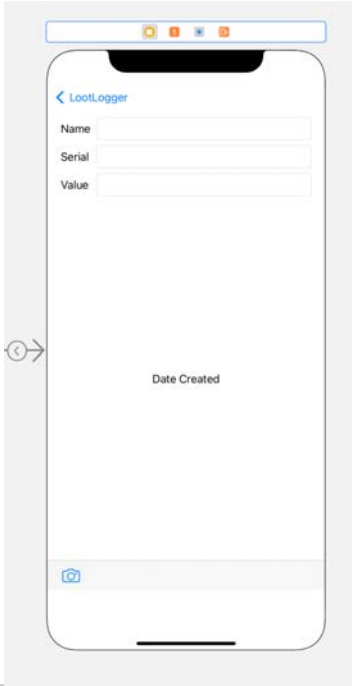




Presenting View Controllers

By default, a new instance of UIToolbar that is created in an interface file comes with one UIBarButtonItem. Select this bar button item and open the attributes inspector. Change the System Item to Camera, and the item will show a camera icon.





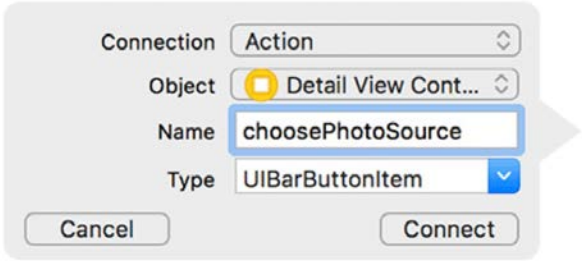
Presenting View Controllers

Build and run the application and navigate to an item's details to see the toolbar with its camera bar button item.

With **Main.storyboard still** open, **Option-click DetailViewController.swift** in the project navigator to open it in another editor.

In Main.storyboard, select the camera button in the document outline and Control-drag from the selected button to the DetailViewController.swift editor.

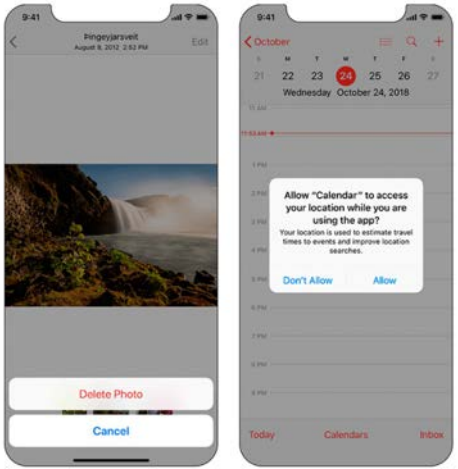
In the panel, select Action as the Connection,
name it choosePhotoSource,
select UIBarButtonItem as the Type, and click Connect



Alerts

To allow the user to choose a photo source, you will present an alert with the possible choices. Alerts are often used to display information the user must act on. When you want to display an alert, you create an instance of **UIAlertController** with a preferred style. The two available styles are UIAlertControllerStyle.actionSheet a

The .actionSheet style is used to present the user with a list of actions to choose from. The .alert type is used to display critical information and requires the user to decide how to proceed. The distinction may seem subtle, but if the user can back out of a decision or if the action is not critical, then an .actionSheet is probably the best choice.



.actionSheet .alert

Alerts

In DetailViewController.swift, update choosePhotoSource(_:) to create an alert controller instance.

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

Alerts

After determining that the user wants to associate a photo with some item, you create an instance of UIAlertController. No title or message are needed for this action sheet since the purpose should be self-evident from the action the user took. Finally, you specify the .actionSheet style for the alert.

If the alert controller were presented with the current code, there would not be any actions for the user to choose from. You need to add actions to the alert controller, and these actions are instances of UIAlertAction. You can add multiple actions (regardless of the alert's style). They are added to the UIAlertController instance using the addAction(_:) method.

Alerts

```
IBAction func choosePhotoSource(_ sender: UIBarButtonItem) {
    let alertController = UIAlertController(title: nil,
                                            message: nil,
                                            preferredStyle: .actionSheet)
    let cameraAction = UIAlertAction(title: "Camera", style: .default) { _ in
        print("Present camera")
    }
    alertController.addAction(cameraAction)

    let photoLibraryAction
            = UIAlertAction(title: "Photo Library", style: .default) { _ in
        print("Present photo library")
    }
    alertController.addAction(photoLibraryAction)

    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
    alertController.addAction(cancelAction)
}
```

Alerts

Each action is given a title, a style, and a closure to execute if that action is selected by the user. The different styles – .default, .cancel, and .destructive – influence the position and styling of the action within the action sheet. For example, .cancel actions show up at the bottom of the list, and .destructive actions use red font colors to emphasize the destructive nature of the action.

Now that the action sheet has been configured, you need a way to present it to the user. To present a view controller modally, you call present(_:animated:completion:) on the initiating view controller, passing in the view controller to present as the first argument.

Alerts

Update choosePhotoSource(_:) to present the alert controller modally:

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

alertController.addAction(photoLibraryAction)

    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
    alertController.addAction(cancelAction)

    present(alertController, animated: true, completion: nil)
}
```

Alerts

The present(_:animated:completion) method takes in a view controller to present, a Bool indicating whether that presentation should be animated, and an optional closure to call once the presentation is completed. Generally, you will want the presentation to be animated, as this provides context to the user about what is happening.

Build and run the application. Tap the camera button and watch the action sheet slide up. Finally, tap one of the actions. If you tap either the Camera or Photo Library action, you will see a message logged to the console indicating which was tapped. Regardless of which action you tap, you will notice that the action sheet is automatically dismissed.

Presentations Styles of View Controllers

.automatic
Presents the view controller using a style chosen by the system. Typically this results in a .formSheet presentation. This is the default presentation style.

.formSheet
Presents the view controller centered on top of the existing content

.fullScreen
Presents the view controller over the entire application.

.overFullScreen
Similar to .fullScreen except the view underneath the presented view controller stays visible. Use this style if the presented view controller has transparency

.popover
Presents the view controller in a popover view on iPad. (On iPhone, using this style falls back to a form sheet presentation style due to space constraints)

Presentations Styles of View Controllers

Action sheets should be presented using the popover style.
on iPad this produces a popover interface with a “pointer” connecting it to the element that triggered it.
On iPhone, because of the smaller window size, .popover falls back to .automatic and allows the system to choose the best style.

This is what you want for your alert controller. On iPad, you want it to appear in a popover pointing at the camera bar button. On iPhone, you want the system to select the best style for the screen size (which will be the .formSheet style you just saw in action).

Presentations Styles of View Controllers

Update choosePhotoSource(_:) to tell the alert controller to use the popover presentation style.

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

    alertController.modalPresentationStyle = .popover
```

Presentations Styles of View Controllers

To indicate where the popover should point, you can specify a frame or a bar button item for it to point to. Since you already have a bar button item, that is the better choice here.

In choosePhotoSource(_:), specify the bar button item that the popover should point at.

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

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

Presentations Styles of View Controllers

Every view controller has a popoverPresentationController, which is an instance of UIPopoverPresentationController. The popover presentation controller is responsible for managing the appearance of the popover. One of its properties is barButtonItem, which tells the popover to point at the provided bar button item. Alternatively, you can specify a sourceView and a sourceRect if the popover is not presented from a bar button item.

Build and run the application in iPad simulator, navigate to an item's details, and tap the camera button. The action sheet is presented in a popover pointing at the camera button (Notice that there is no "cancel" action; when an action sheet is presented in a popover, the cancel action is triggered by tapping outside of the popover.

Presentations Styles of View Controllers

Every view controller has a popoverPresentationController, which is an instance of UIPopoverPresentationController. The popover presentation controller is responsible for managing the appearance of the popover. One of its properties is barButtonItem, which tells the popover to point at the provided bar button item. Alternatively, you can specify a sourceView and a sourceRect if the popover is not presented from a bar button item.

Build and run the application in iPad simulator, navigate to an item's details, and tap the camera button. The action sheet is presented in a popover pointing at the camera button (Notice that there is no "cancel" action; when an action sheet is presented in a popover, the cancel action is triggered by tapping outside of the popover.