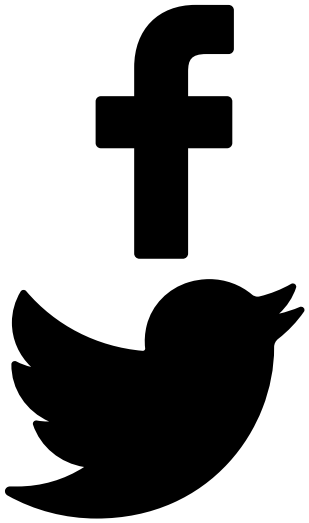
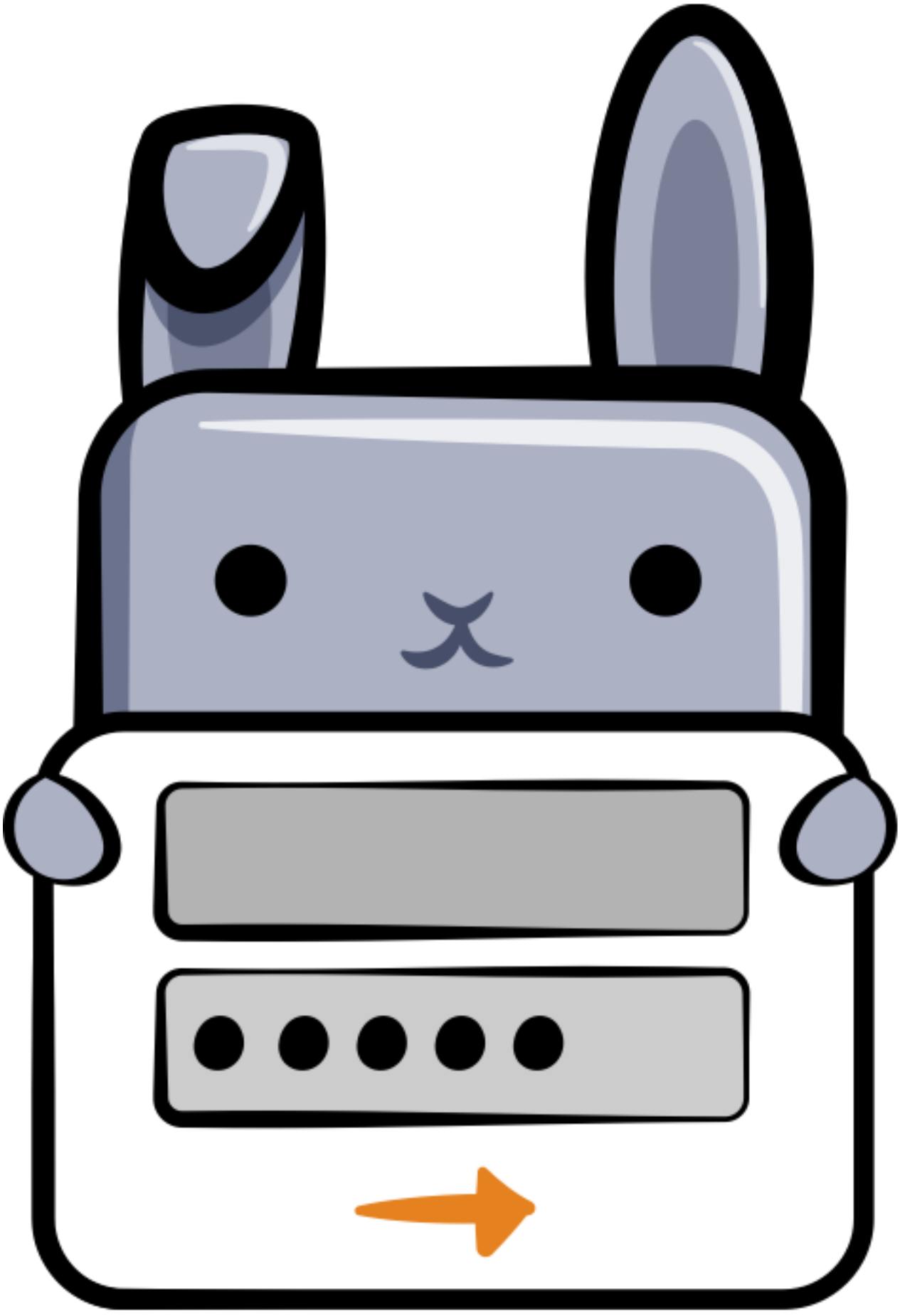




Get Started

Adopting Scenes in iPadOS

In this Adopting Scenes tutorial, you'll learn how to support multiple windows by creating a new iPadOS app. You'll also learn how to update existing iOS 12 apps to support multiple windows.

By Mark Struzinski Dec 16 2019 · Article (30 mins) · Intermediate

5/5

5 Ratings

iPadOS recently debuted alongside iOS 13.1 and adopts an exciting new feature available for the first time: window support!

Window support allows users to create multiple windows of your running app. Windows can contain similar content, or a new window can contain a completely different view, such as accessory information.

In this tutorial, you'll learn how to:

Update your existing apps to include scene support.

Interact with the new set of objects in the app delegate to support multiple windows.

Create new windows and update them whether they're in the foreground or background.

Before getting started, make sure to click the *Download Materials* button at the top or bottom of this tutorial.

Getting Started

There are some fundamental changes to the application lifecycle with iPadOS and iOS 13. Windows are now managed by a `UISceneSession`.

There is a new set of objects to manage the window lifecycle: `UISceneSession`, `UISceneDelegate`, and `UISceneConfiguration`. Scenes manage windows, and they have their own dedicated lifecycle managed outside of the `UIApplication` instance.

These changes to the components also change the way you interact with `UIApplicationDelegate`. A lot of the delegate callbacks used in iOS 12 have now moved to `UISceneSessionDelegate`.

Scene Components

There are several new objects that you'll have to interact with to support multiple scenes.

UIWindowScene

`UIWindowScene` is a subclass of `UIScene` and the most common type of scene you'll interact with. This object represents one instance of your app's user interface. You shouldn't instantiate `UIWindowScene` directly; UIKit will create it for you. In most cases, you'll interact with `UIScene` through its delegate, `UISceneSessionDelegate`.

UIWindowSceneDelegate

`UIWindowSceneDelegate` is a subclass of `UISceneDelegate` and contains the core methods you'll use to respond to `UIWindowScene` lifecycle events.

You don't instantiate this object directly. UIKit creates it for you and pairs it with each `UIWindowScene`. Your *Info.plist* dictates how UIKit creates the scene delegate. You'll learn how to set this up in a later section.

UISceneConfiguration

This object contains information that tells UIKit how to instantiate your scene. You configure a `UISceneConfiguration` in code or in your *Info.plist*.

Apple's preferred method is to use the *Info.plist*, and this tutorial will also use the plist file for scene

configuration.

UISceneSession

This object contains information about an app's scene. It manages the lifecycle of exactly one `UIScene` and contains persisted interface states. The operating system creates scene sessions. You shouldn't instantiate them directly. The graph below shows the relationship between some of these objects.

iPadOS and Multiple Scenes

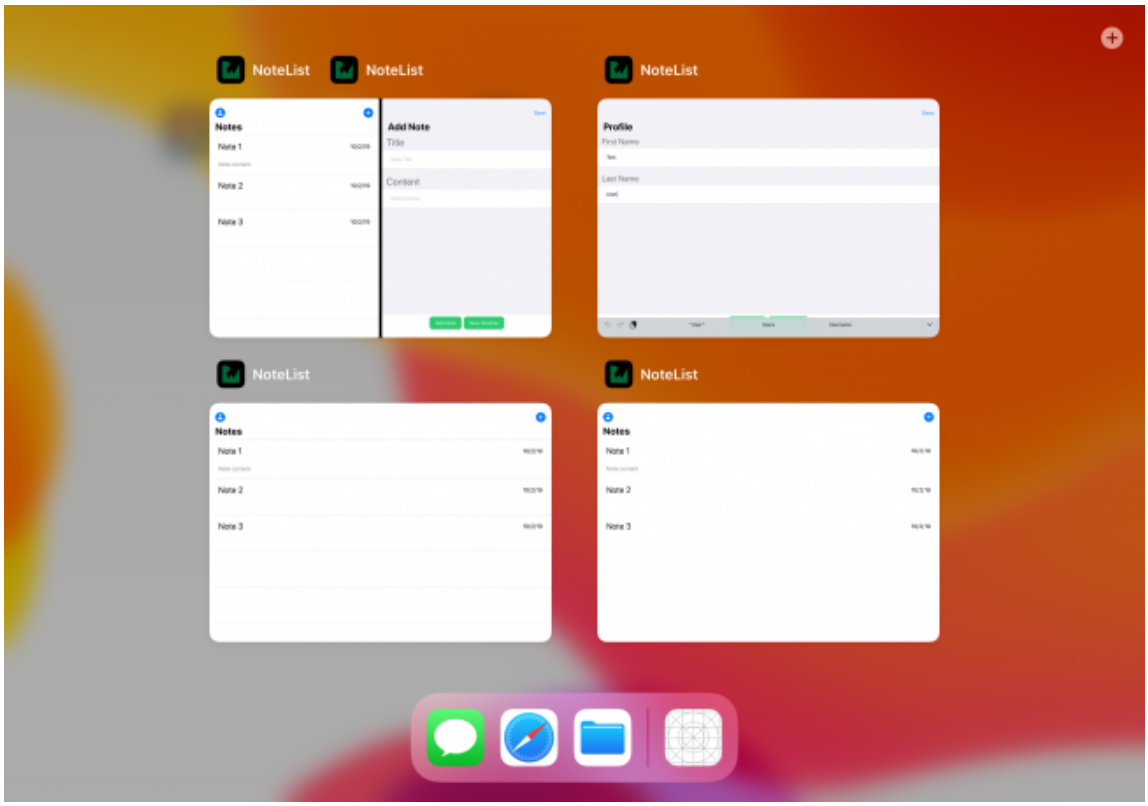
iPadOS adds several new user interactions related to scene support. Time to explore how users employ these new interactions.

Creating New Windows

A user creates new windows using several different methods:

Through the App Exposé Multitasking View

If your app declares multiple-window support, your user can enter multitasking view by swiping up to show the dock and on your app icon in the dock while the app is already running. This shows the following view of your app's windows, including a plus button to create a new window:



Via an Explicit User Interaction

You should only create new windows in reaction to explicit user interaction, such as the user tapping a *New Window* button. This allows your user to control windows and window management. You create a new window programmatically using a new API on `UIApplication`.

Via Drag and Drop

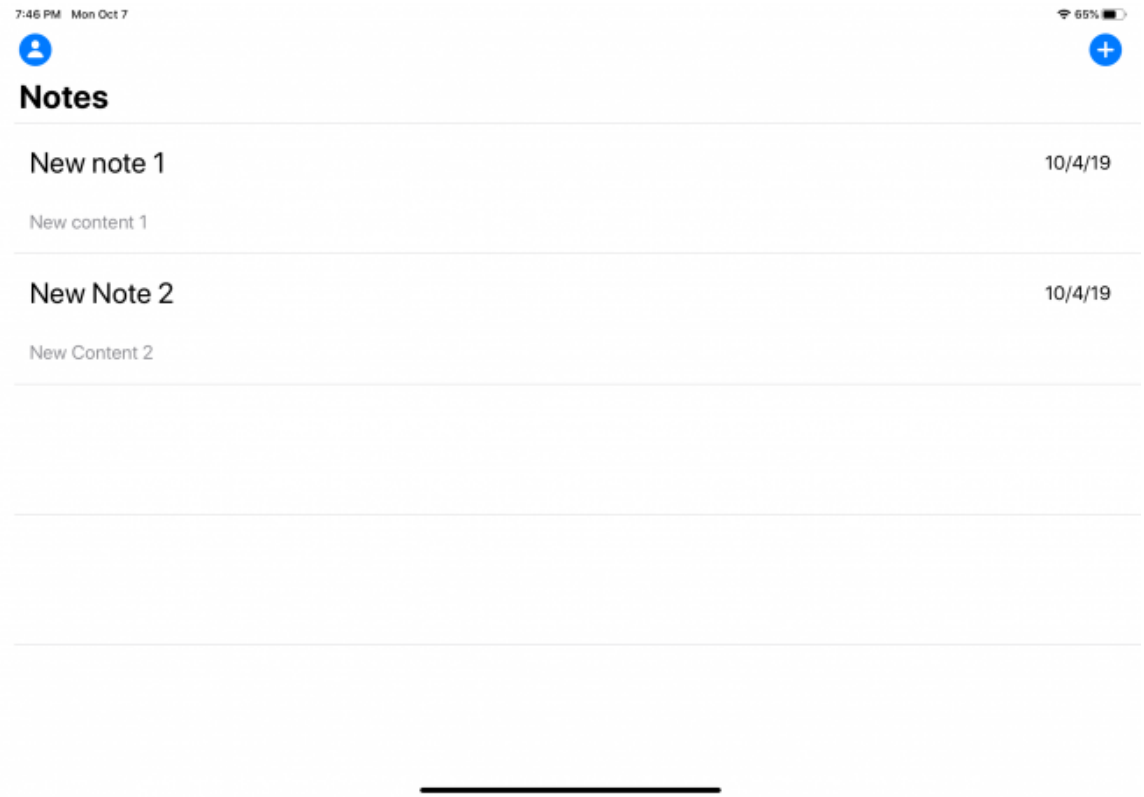
A drag and drop session can instantiate a new scene. The use case for this interaction is dragging an object from your app to expose a new view with detail.

For example, a user drags a list item out of the list, and it instantiates a new window side-by-side with the existing list view. The new window content contains the detail view of the item.

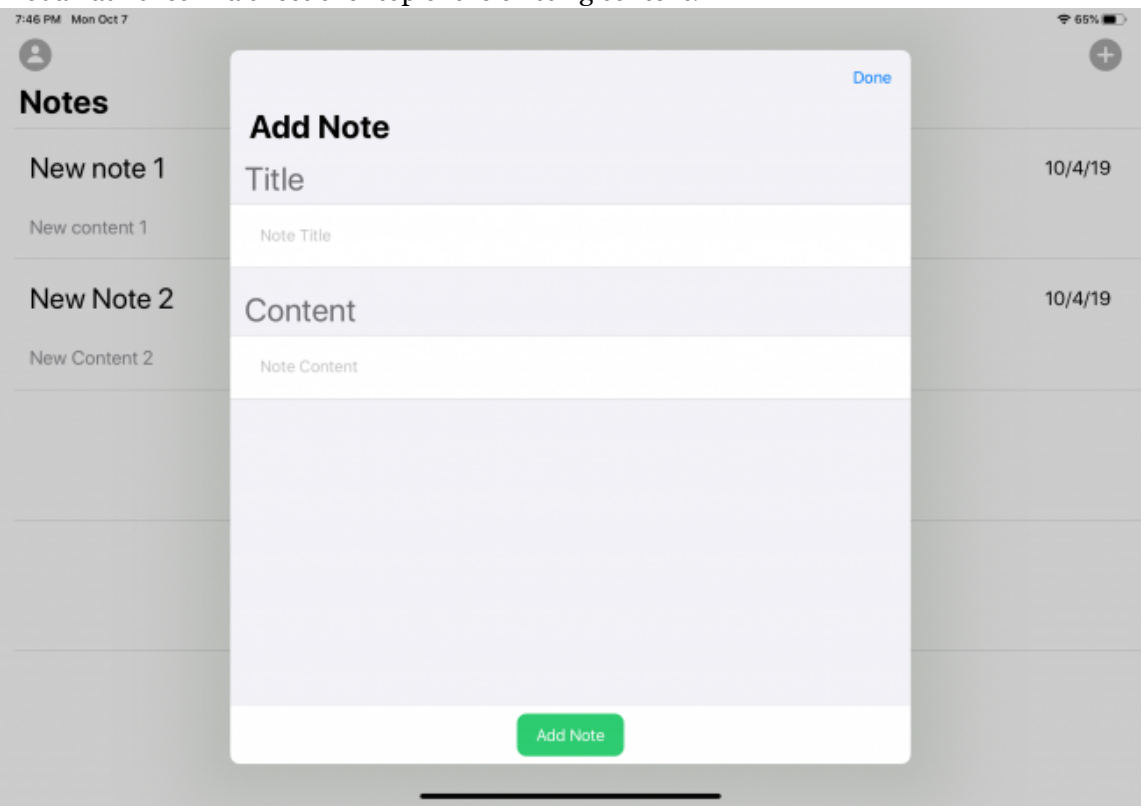
The Example App

If you haven't already, download the example app and open it in the begin folder. Build and run it on an iPad simulator to check it out — it's a simple app that allows you to create notes, view them in a list and delete them. You can also view and update your profile.

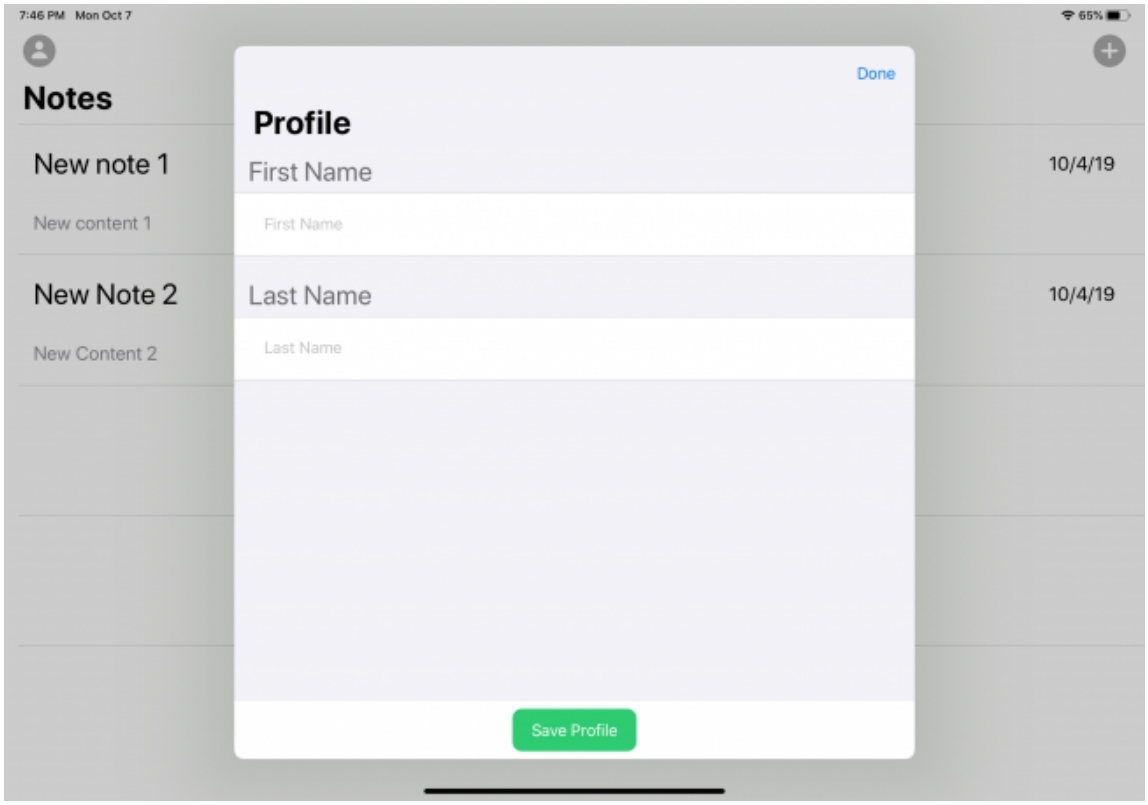
The starter version of the app does not support multiple windows. You can only launch one instance of the app at a time.



This is a SwiftUI app using Core Data to persist notes and user defaults to store profile information. This allows you to work on multiple-window support without worrying about backing storage, network requests or sync issues.
Add a few notes by tapping the plus button in the navigation bar in the top-right. You'll see the new note modal launches in a sheet over top of the existing content:



Add or update some profile information by tapping the profile button in the navigation bar on the top-left. The profile modal displays in a sheet:



The Plan

The rest of this tutorial will teach you how to implement multiple windows in an iOS 13 app.
You will:
Display multiple windows.
Add new scene types.
Sync data across windows, whether they are in the foreground or background.

Adding Multiple Scene Support

First, you'll add support for multiple scenes to your app. If you created your app with Xcode 11, then Xcode set up the scene delegate and scene configuration for you. Xcode also handles the connection between app delegate and scene delegate in the default iOS 13 template.
If you are upgrading an existing app from iOS 12, follow the steps in the *Updating Your App from iOS 12* section before you get started.

Update the Info.plist

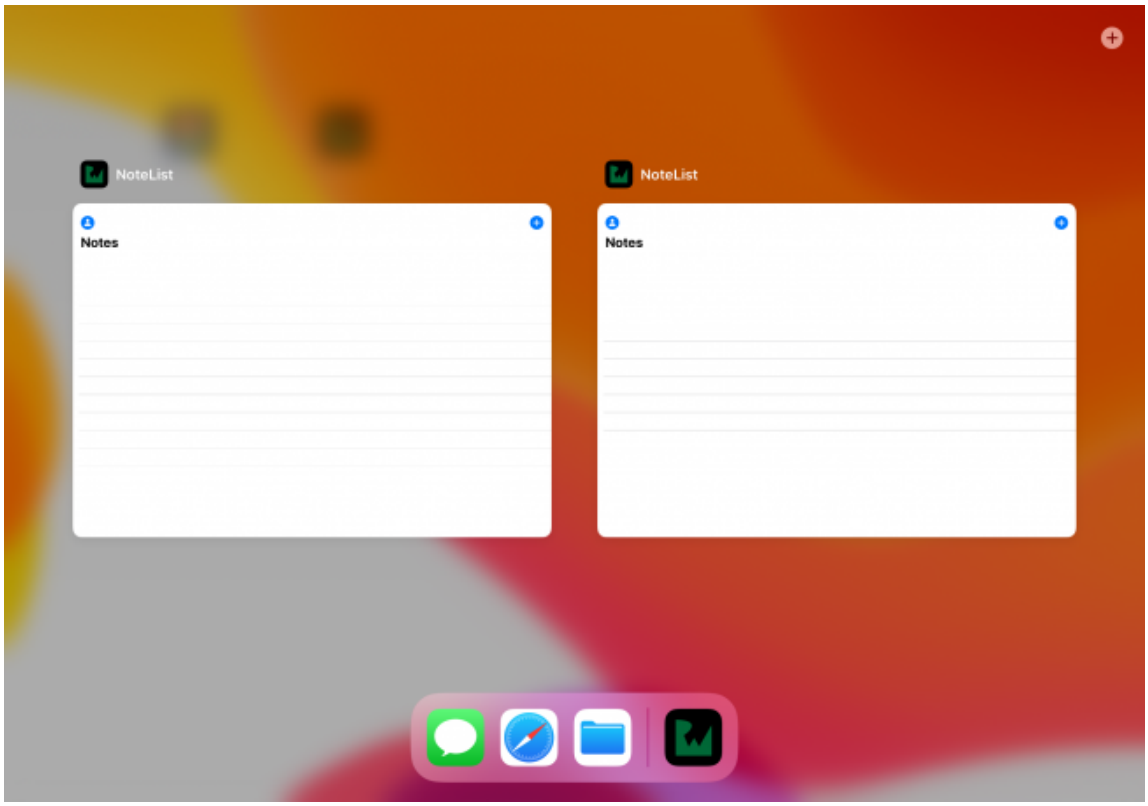
First, update *Info.plist* to support multiple scenes:

. Open *Info.plist*.

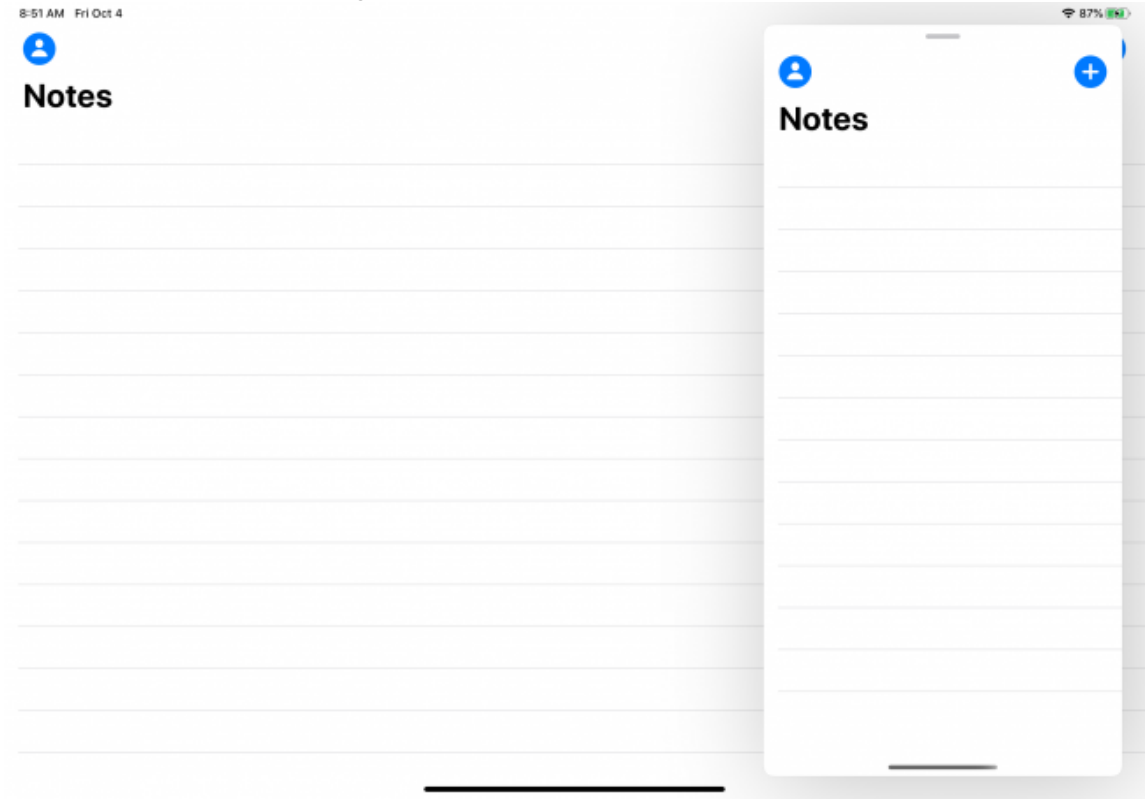
. Expand the *Application Scene Manifest* node.

. Set the value of *Enable Multiple Windows* to *YES*.

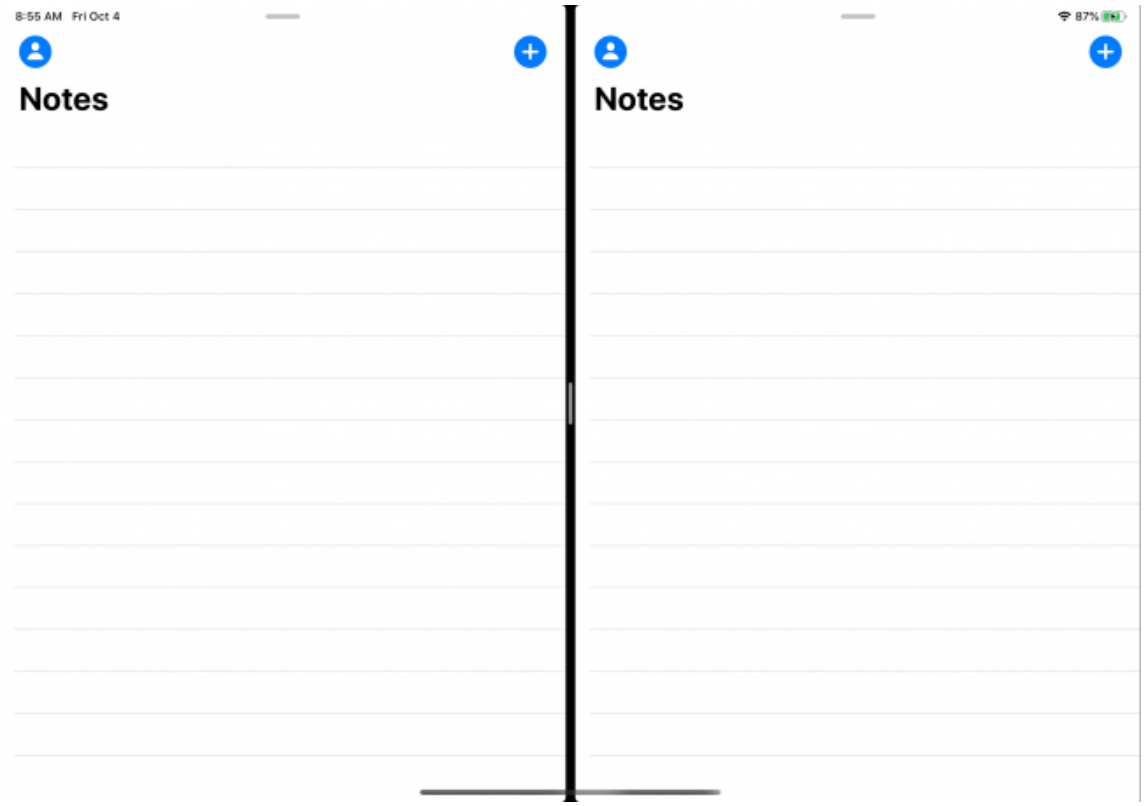
After you have set this value, you can launch multiple windows of the same scene.
Build and run now on an iPad simulator.
Switch to landscape to give yourself more room. You can now perform the following actions with the app:
While the app is in the foreground, slide up from the bottom bezel to launch the dock. Tap and hold the right-hand app icon. Select *Show All Windows* from the menu. You'll now see the App Exposé view and can tap the plus icon on the top right to launch a new window:



With the app in the foreground, slide up to show the dock again. This time, drag the app icon out of the dock until it becomes a hovering window. Drop the window on the right or left side of the screen. You now have a second window running in slideover mode:



With the slideover window still running, tap and hold on the drag handle at the top of the window. Pull down and to the right until the window changes shape. Drop the window onto the side of the screen. Now you have two windows running side-by-side. You can also move the split handle in the middle to resize these windows:



That's a lot of support for updating one value in the plist! Moving forward, adding support for additional scenes is not so simple. You have to consider where it makes sense to add support and how to keep your scenes in sync.

Adding a New Scene

In the current state, when you tap the *New Note* button, a modal sheet opens to enter a new note. After you dismiss the window, the new note shows up in the list.

What if a user wants to enter new notes in one window and watch the list refresh in a different one side-by-side? This is a valid use case and increases the usefulness of the app. But how, you ask, do you add support for a brand new scene?

Create a New Scene Delegate

First, you need to add a new scene delegate to respond to events for the new scene. Under the *App* group in Xcode, add a new Swift file by selecting *File ▸ New ▸ File* and picking the *Swift File* template. Name the new file *CreateDelegate.swift* and click *Create*.

Add the following code to the new file:

```
// 1
import SwiftUI

// 2
class CreateDelegate: UIResponder, UIWindowSceneDelegate {
  // 3
  var window: UIWindow?

  // 4
  func scene(
    _ scene: UIScene,
    willConnectTo session: UISceneSession,
    options connectionOptions: UIScene.ConnectionOptions
  ) {
    if let windowScene = scene as? UIWindowScene {
      // 5
      let window = UIWindow(windowScene: windowScene)
```

```
        // 6
        window.rootViewController = UIHostingController(
          rootView: AddNoteView(addNotePresented: .constant(false))
        )
        // 7
        self.window = window
        // 8
        window.makeKeyAndVisible()
      }
    }
  }
```

With this code, you:

. Import SwiftUI, since you need to use a hosting view and invoke a SwiftUI view here.

. Declare conformance to `UIWindowSceneDelegate`, allowing you to respond to window scene events.

. Create a variable to hold a `UIWindow`. You populate this when you create your scene.

. Implement `scene(_:willConnectTo:options:)`, which allows you to define the startup environment and views.

. Create a new window using the `UIWindowScene` passed to you by UIKit.

. Instantiate a new instance of `AddNoteView` and set it as the root view of a new `UIHostingController`. Since it's not running in a modal context, pass `false` for `addNotePresented` argument.

. Set the `window` property to the new window.

. Make the new window visible.

Add a New Scene to the Scene Manifest

Next, you need to declare support for the new scene in the scene manifest. This is where you'll declare your new scene and tell UIKit where to get the scene delegate for it. To do so:

. Open *Info.plist* again.

. Expand *Application Scene Manifest*.

. Open *Scene Configuration* ▸ *Application Session Role* nodes.

. Tap the plus (*+*) button next to *Application Session Role* to add a new entry.

. Drag this entry underneath the *Default Configuration* entry.

. Expand the new entry and delete the *Class Name* and *Storyboard Name* entries. You'll use the default for the class, which is `UIWindowScene`. There is no need to customize this. Since you're invoking a SwiftUI view, there is no need to declare a storyboard name.

. Enter the value *$(PRODUCT_MODULE_NAME).CreateDelegate* for *Delegate Class Name*. This tells UIKit to use your new `CreateDelegate` when initializing the new scene from the main target module.

. Enter the value *Create Configuration* for *Configuration Name*. This is the name UIKit will use to look up configuration setup to create your new scene.

Your *Application Scene Manifest* entry should now look like this:

▼ Application Scene Manifest	Dictionary	(2 items)
Enable Multiple Windows	Boolean	YES
▼ Scene Configuration	Dictionary	(1 item)
▼ Application Session Role	Array	(2 items)
▼ Item 0 (Default Configuration)	Dictionary	(3 items)
UILaunchStoryboardN...	String	LaunchScreen
Configuration Name	String	Default Configuration
Delegate Class Name	String	$(PRODUCT_MODULE_NAME).SceneDelegate
▼ Item 1 (Create Configuration)	Dictionary	(2 items)
Delegate Class Name	String	$(PRODUCT_MODULE_NAME).CreateDelegate
Configuration Name	String	Create Configuration

Add UI to Display the New Scene

You've declared support for the new scene, but now you need a way to display it.
To do so, add a button to the *New Note* view that will allow the user to make the modal a new window. This will create a new scene and place it side by side with the existing note list scene.
UIKit always calls `application(_:configurationForConnecting:options:)` in your `UIApplicationDelegate` class to determine which scene configuration to use when bootstrapping a new scene.
This is where you customize the startup experience. Hooking into the `NSUserActivity` system is how you specify scene creation options.
When UIKit creates the new scene, it will pass any existing `NSUserActivity` objects into the `UIScene.connectionOptions` parameter of this method. You can use that activity to inform UIKit which scene configuration to use.

Create a Constant Declaration for Scene Names

First, select *NoteList* group in the Project navigator. Create a new group using *File ▸ New ▸ Group* and name it *Constants*. Select *File ▸ New ▸ File* and pick the *Swift File* template. Name the new file *SceneConfigurationNames.swift*, make sure *Constants* group is selected as the destination, and click *Create*.
Add the following code underneath `import Foundation`:

```
struct SceneConfigurationNames {
  static let standard = "Default Configuration"
  static let create = "Create Configuration"
}
```

This creates some constants you can use to reference configuration names.

Create an enum Declaration for User Activities

To create an enum to reference user activities, select *File ▸ New ▸ File* and pick the *Swift File* template. Name the file *ActivityIdentifier.swift*, select the *Constants* group as the destination, and click *Create*.
Enter the following code:

```
import UIKit

enum ActivityIdentifier: String {
  case list = "com.raywenderlich.notelist.list"
  case create = "com.raywenderlich.notelist.create"

  func sceneConfiguration() -> UISceneConfiguration {
    switch self {
    case .create:
      return UISceneConfiguration(
        name: SceneConfigurationNames.create,
        sessionRole: .windowApplication
      )
```

```
    case .list:
      return UISceneConfiguration(
        name: SceneConfigurationNames.standard,
        sessionRole: .windowApplication
      )
    }
  }
}
```

This specifies an easy-to-use and type-safe enum that you can use to identify scenes via `NSUserActivity`. This prevents the need to pass string literals around when referencing *NSUserActivity* identifiers. It also creates a simple convenience method to generate a scene configuration from a given activity identifier.

Add UI to Create a New Window

Now you can connect all of this together by adding a *New Window* button to the *Add Note* view. Open *AddNoteFormButtonView.swift*. Inside the `HStack`, right under the first button declaration, add the following code:

```
// 1
if UIDevice.current.userInterfaceIdiom == .pad {
  // 2
  Button("New Window") {
    // 3
    let userActivity = NSUserActivity(
      activityType: ActivityIdentifier.create.rawValue
    )
    // 4
    UIApplication
      .shared
      .requestSceneSessionActivation(
        nil,
        userActivity: userActivity,
        options: nil,
        errorHandler: nil)
    // 5
    self.addNotePresented = false
  }.padding(EdgeInsets(top: 12, leading: 20, bottom: 12, trailing: 20))
    .foregroundColor(Color.white)
    .background(Color(red: 46/255, green: 204/255, blue: 113/255))
    .cornerRadius(10)
}
```

Here is what this code does:

. Only runs on an iPad. iPhone devices do not currently support multiple scenes.

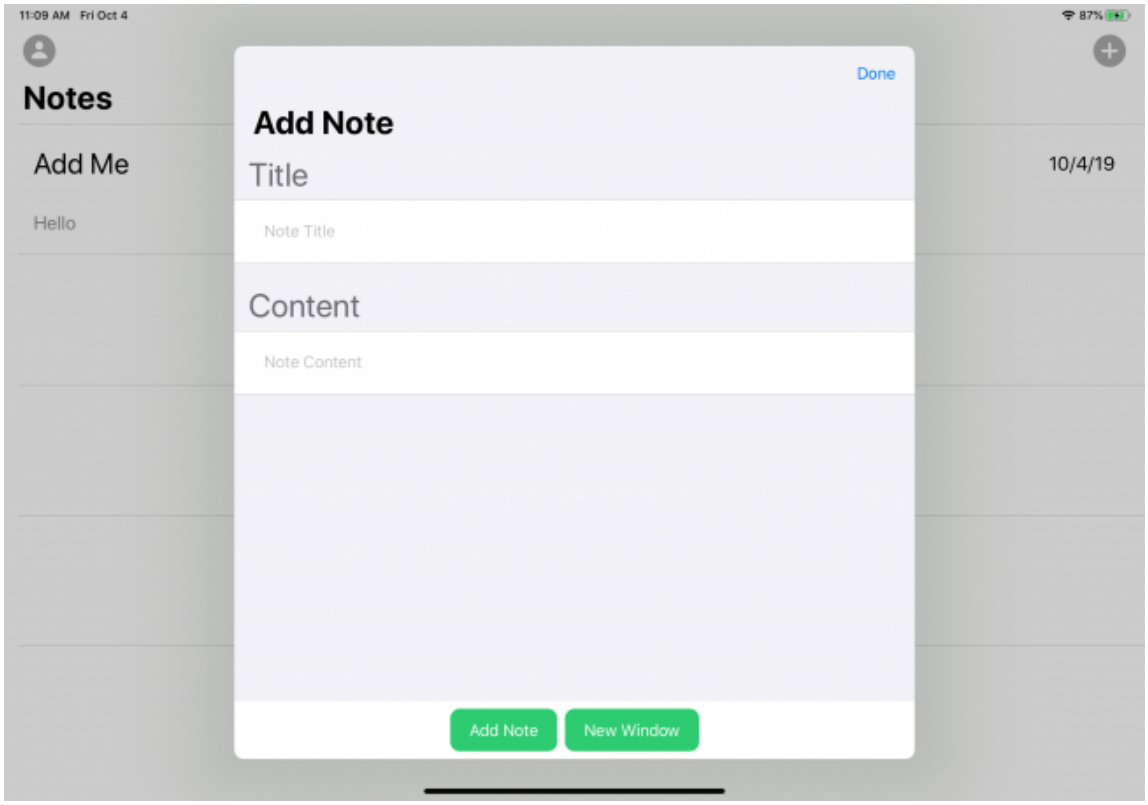
. Creates a new button with the title *New Window* and adds some styling to it.

. Instantiates a user activity with the enum you created earlier in this section.

. Requests a new scene session from `UIApplication` and passes it the user activity you created in the previous section.

. Sets the binding variable `addNotePresented` to `false`, which will tell the note list view to dismiss the modal.

Build and run. Tap the plus button on the top-right to check out the new button on the bottom of the *Add Note* view:



Update the App Delegate to Configure the New Scene

Finally, you need to update the app delegate to use the new configuration for the *create* user activity if it's in the connection options.

Open *AppDelegate.swift*, find and replace `application(_:configurationForConnecting:options:)` with this:

```
func application(
  _ application: UIApplication,
  configurationForConnecting connectingSceneSession: UISceneSession,
  options: UIScene.ConnectionOptions)
    -> UISceneConfiguration {
  // 1
  var currentActivity: ActivityIdentifier?

  // 2
  options.userActivities.forEach { activity in
    currentActivity = ActivityIdentifier(rawValue: activity.activityType)
  }

  // 3
  let activity = currentActivity ?? ActivityIdentifier.list

  // 4
  let sceneConfig = activity.sceneConfiguration()

  // 5
  return sceneConfig
}
```

Here, you:

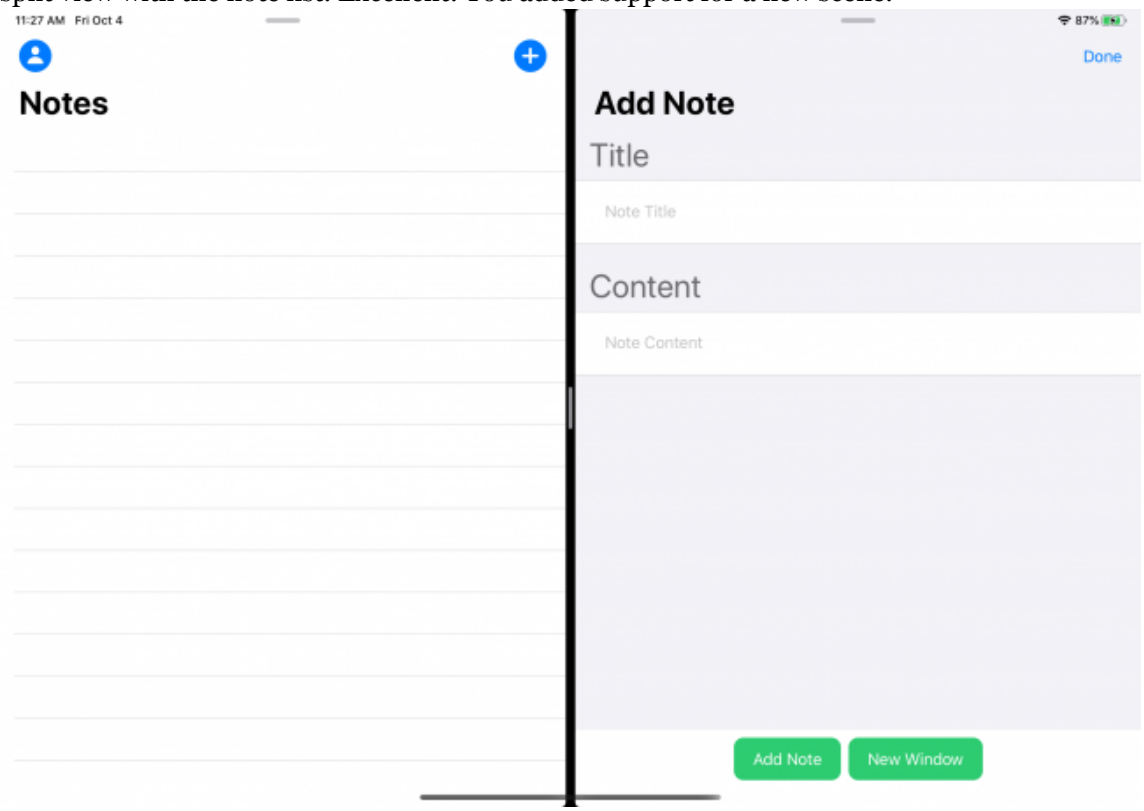
. Create a variable to hold the current user activity.

. Check the connection options for user activities and attempt to generate an `ActivityIdentifier`.

. Use the activity if found. If not, default to list.

. Create a new scene configuration by using the convenience method on `ActivityIdentifier`.

. Return the new scene configuration.
Build and run one more time.
Attempt to add a new note, then tap the *New Window* button. This time, a new window will open in split view with the note list. Excellent! You added support for a new scene.



Keeping Scenes up to Date

Even though you've implemented multiple windows and scenes, you may notice there is a problem with the state of the UI when the data changes.
Open a note list window and an Add Note window side-by-side. When you add a new note, you can see that the changes don't update to the list window.
If you stop and restart the app, you can see the list updates with the latest data, which means the notes are present in the Core Data store. You need a way of telling the UI to update its current state and re-fetch data. This is where `UISceneSession` comes into play. A scene session can be in one of the following states:
Foreground Active: The scene is running in the foreground and currently receiving events.
Foreground Inactive: The scene is running in the foreground but not currently receiving events.
Background: The scene is running in the background and is not on screen.
Unattached: The scene is not currently connected to your app.
Scenes can become disconnected at indeterminate times. The operating system may disconnect scenes at any point to free up resources.
You have to handle both foreground and background scenarios to keep your scenes up-to-date with their backing data.

Keeping Foreground Scenes up to Date

You'll handle foreground sessions with a familiar tool: *NotificationCenter*.
You can refresh any foreground sessions by listening for the appropriate notification and requesting updates from the Core Data store. You'll do so by adding a typed notification name for refresh scenarios.
Create a new group in Xcode and name it *Extensions*. Select *File ▸ New ▸ File* and pick the *Swift File* template. Add a new file and name it *Notification+Name*.
Add the following content to it:

```
extension Notification.Name {
  static let noteCreated = Notification.Name("com.raywenderlich.notecreated")
}
```

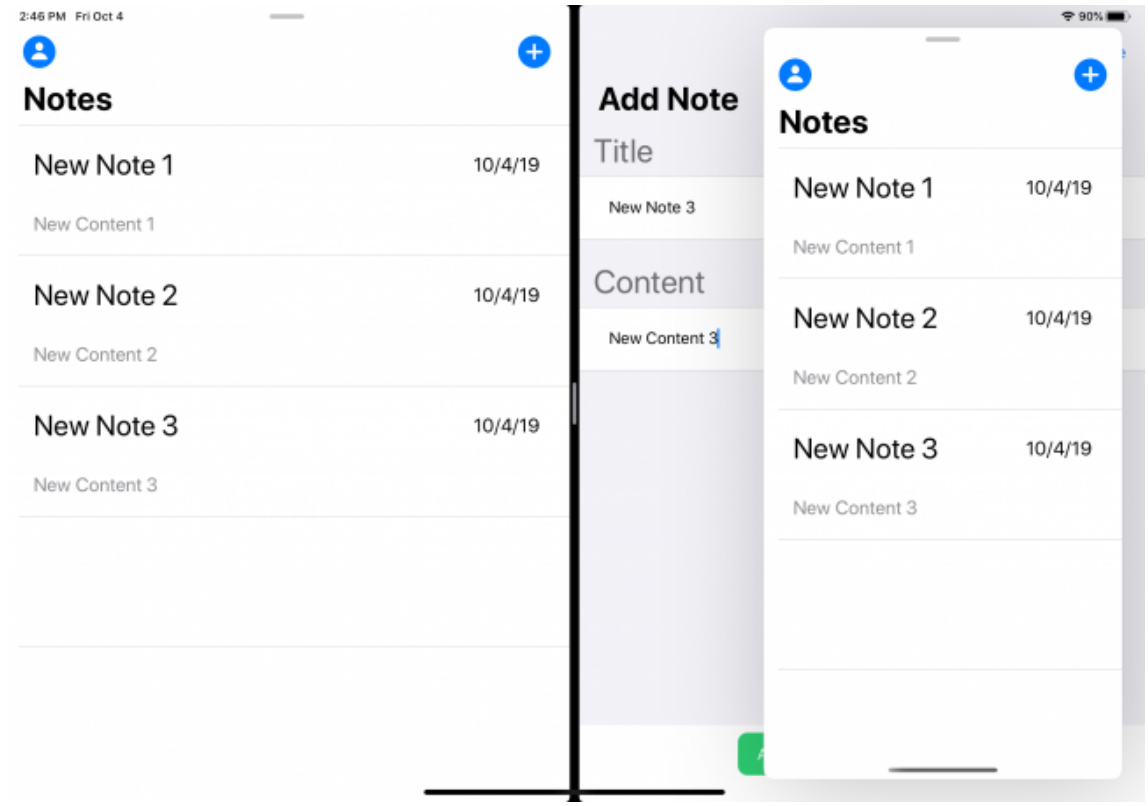
This creates a typed notification name for reuse.
You'll post a notification for any new notes. Open *AddNoteViewModel.swift*. and find `createNote()`.
Add the following line after the request to Core Data to create the new note:

```
NotificationCenter.default.post(name: Notification.Name.noteCreated, object: nil)
```

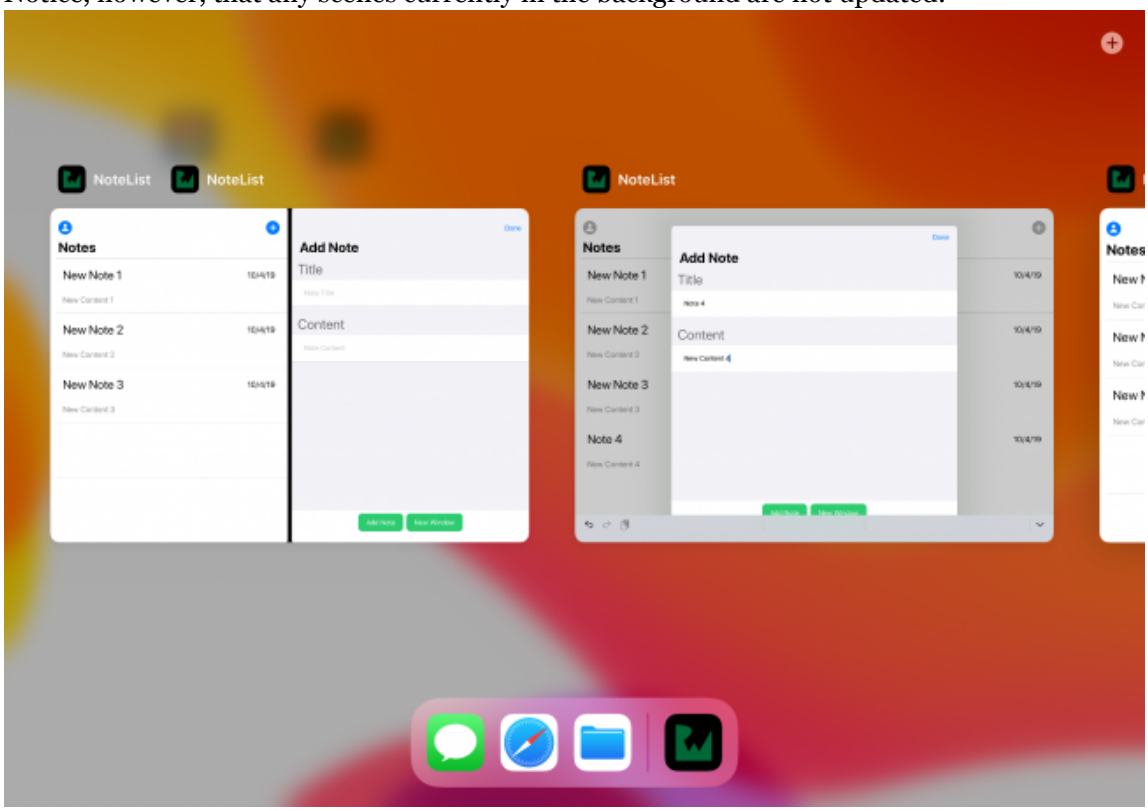
Now every time you create a note, you also post a notification.
You also need to handle the notification in the list view. Open *NoteListViewModel.swift* and add the following method right underneath the property declarations:

```
init() {
  NotificationCenter
    .default
    .addObserver(
      self,
      selector: #selector(handleNoteCreated),
      name: Notification.Name.noteCreated,
      object: nil
  )
}
```

Every list that is on screen has a backing view model. Now each view model will respond to the new note notification by performing a fetch. Since the list views are all bound to the `notes` array, they will update every time the data changes.
Build and run.
Put a new note view side-by-side with a note list view. Next, add several notes, and you can see the note list update in real-time.
Then, add a third list window in slideover mode. Continue adding notes. Now both list views are updating in real-time.



Notice, however, that any scenes currently in the background are not updated:



You have to use a different API to update the snapshot for backgrounded or disconnected.

Keeping Background Scenes up to Date

A scene that is in the multitasking switcher is backgrounded. It is still possible to update scene snapshots for scenes in this state of the lifecycle.

To find and update these scenes, you first have to attach some identifying information to them. In this way, you can query them later and call update on the ones that need it.

For this purpose, you'll use the `userInfo` dictionary property on the scene session.

Attaching Identifying Information to Scenes

Update `application(_:configurationForConnecting:options:)` in *AppDelegate.swift* to attach a `userInfo` dictionary to the scene session. Right after you create the scene configuration, add the following code:

```
// 1
let userInfo = [
  "type": activity.rawValue
]

// 2
connectingSceneSession.userInfo = userInfo
```

With this code, you:

. Create a user info dictionary from the current activity.

. Set the `userInfo` property on the scene session.

Updating Scenes from the Background

Next, locate and request updates to scenes that are currently in the background. This API will only affect scenes in the background.

Open *AddNoteViewModel.swift* and add the following method after `createNote()`:

```
func updateListViews() {
  // 1
  let scenes = UIApplication.shared.connectedScenes

  // 2
  let filteredScenes = scenes.filter { scene in
    guard
      let userInfo = scene.session.userInfo,
      let activityType = userInfo["type"] as? String,
      activityType == ActivityIdentifier.list.rawValue
      else {
        return false
    }

    return true
  }

  // 3
  filteredScenes.forEach { scene in
    UIApplication.shared.requestSceneSessionRefresh(scene.session)
  }
}
```

Here is what you do above:

. Request all connected scenes from `UIApplication`.

. Filter all scenes to look for the information you attached to the `userInfo` object earlier.

. Ask UIKit to refresh all scenes from the filtered list.

Finally, add the following to the end of `createNote()`:

```
updateListViews()
```

To see this in action, you'll need to delete the app from the simulator and build and run again. This is because the older scenes won't have the correct `userInfo` dictionaries set, and they won't refresh properly. Now build and run. You should be able to background a few sessions, create notes in the

foreground and see the background snapshots update when you go into App Exposé.

Updating Your App From iOS 12

This section gives you the steps to update an older project, but it doesn't include a sample project. There are not many steps to adopt basic multiple-window support when updating from iOS 12:
Update your *Info.plist*.
Add a scene delegate.
Update the scene delegate.

Update Info.plist

Find and open the *Info.plist* of your iOS 12 app target and perform the following steps:
. Click the plus button (+) to add a new entry.

. Select *Application Scene Manifest*.

. Open up the new list item by clicking the disclosure triangle (▼).

. Change the value for *Enable Multiple Windows* from *NO* to *YES*.

. Under *Scene Configuration*, click the plus button (+) to add a new scene configuration.

. Select *Application Session Role*.

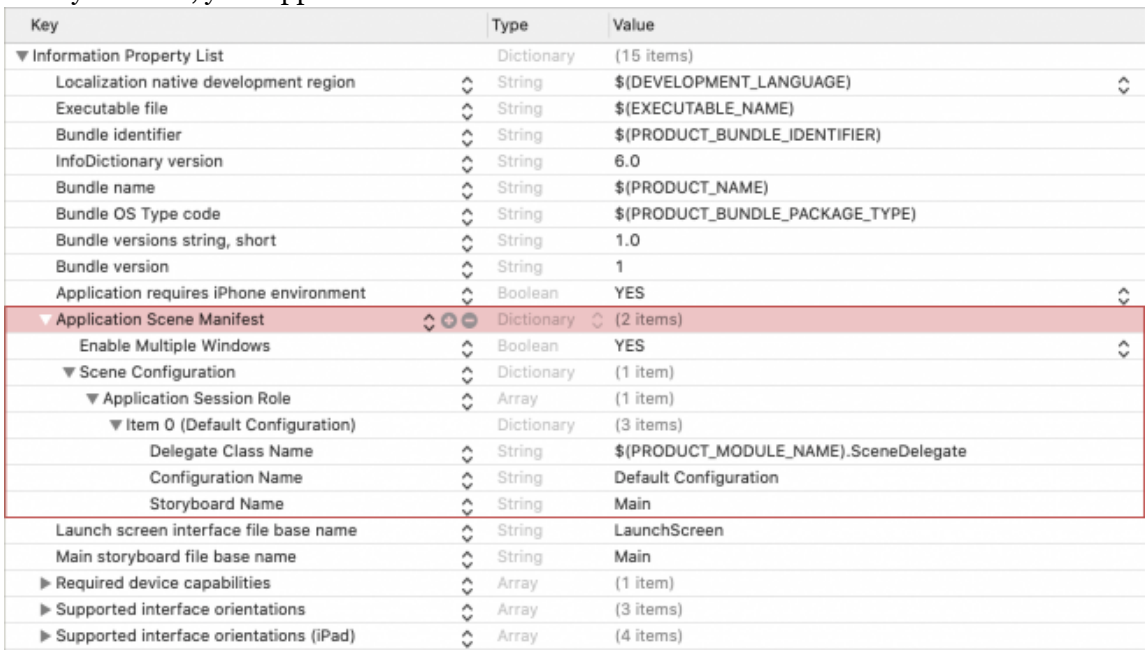
. Click the disclosure triangle to open *Item 0*.

. Enter the following values under each entry:
. *Storyboard Name*: The name of your initial storyboard.

. *Delegate Class Name*: `$(PRODUCT_MODULE_NAME).SceneDelegate`.

. *Configuration Name*: `Default Configuration`.

. You can delete the *Delegate Class Name* entry.
After you finish, your application Scene Manifest should look like this:



Add a Scene Delegate

Next, you need to add a scene delegate. The class name of this scene delegate needs to match the class name you specified for the *Delegate Class Name* entry in the *Info.plist* file the previous step.
Create a new Swift file named *SceneDelegate.swift*, and add the following code to the new file:

```
import UIKit

// 1
@available(iOS 13.0, *)
```

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {
  // 2
  var window: UIWindow?
}
```

Here is what you entered above:

. Since you need to support iOS 12, wrap this class in an availability check. It will only compile for iOS 13.

. The operating system populates `window`, but it has to be present.

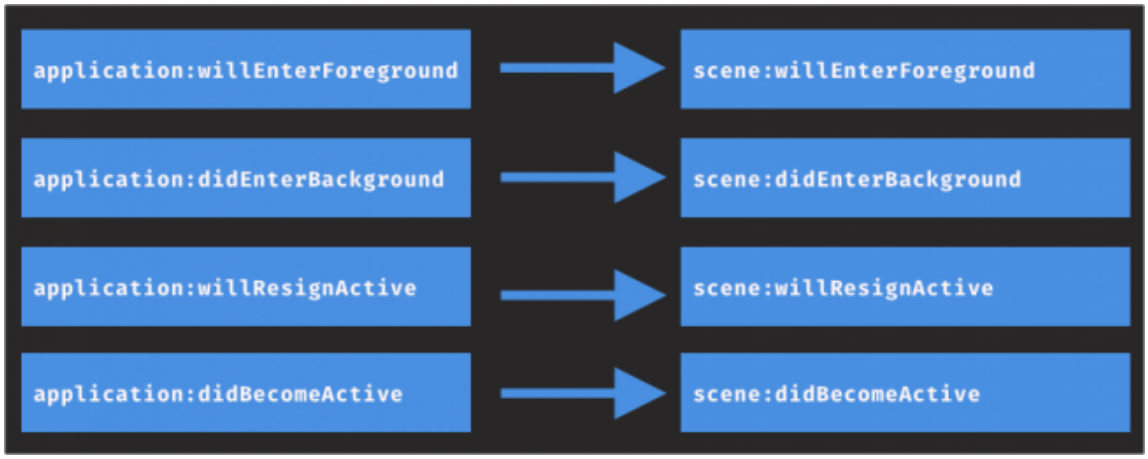
Since you don't need to customize the scene configuration at runtime for this simple example, you don't need to override anything else in this class related to window or session support.

Update the Scene Delegate

You may need to reproduce some of your existing application logic in your scene delegate. In iOS 13, the scene delegate now performs a lot of the functionality that the application delegate used to.

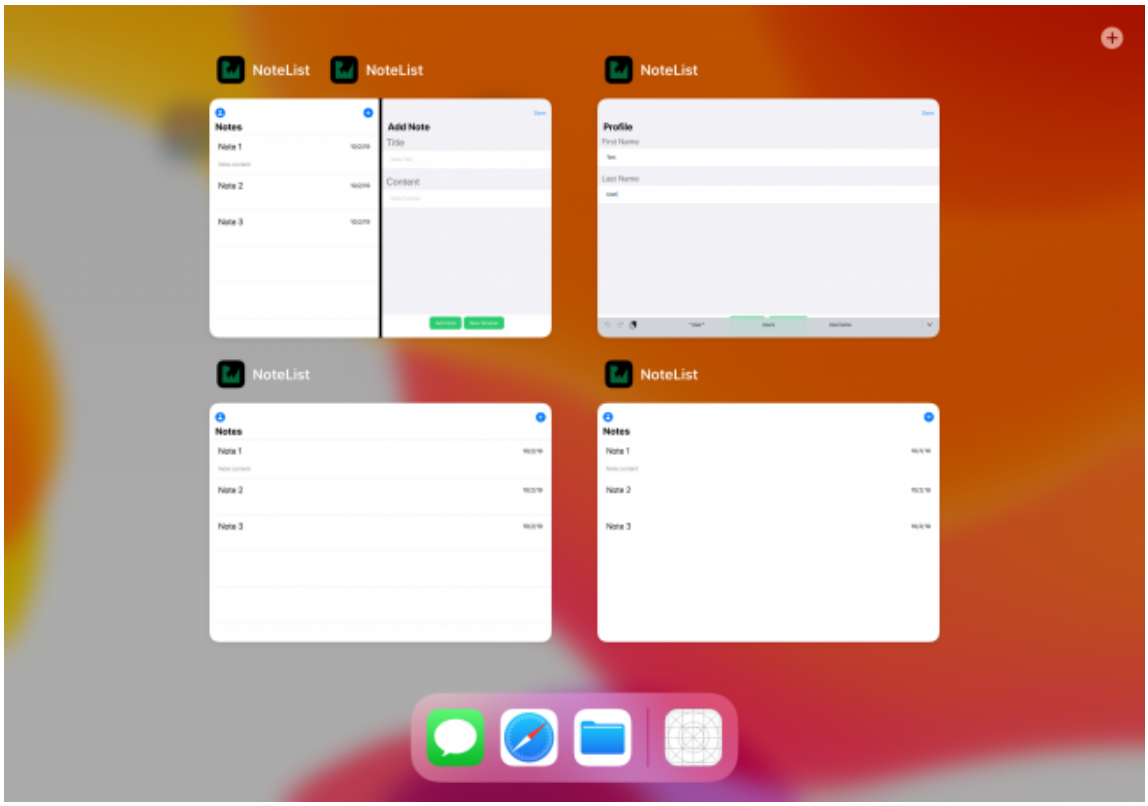
If you are only supporting iOS 13 and up, you'll want to move this logic. If you are supporting iOS 13 and older operating systems, leave your app delegate code as-is and add it to the scene delegate.

The following methods map one-to-one from `UIApplicationDelegate` to `UISceneDelegate`:



Now it's finally time to run your app! If you build and run with an iOS 13 device or simulator, you should be able to create new windows using App Exposé or by dragging out of the dock.

Any new windows should show up in the multitasking view of App Exposé as well.



Now you are ready to move into the next step: Supporting multiple scenes and maintaining state between them.

Where to Go From Here?

You can download the completed version of the project using the *Download Materials* button at the top or bottom of this tutorial.

You learned the main components that compose the new scene APIs and how to upgrade an existing app for multiple scene support.

You also learned how and when to create new scenes and how to update content in foreground and background scenes.

There is still a lot to learn about multiple scene support not covered in this tutorial. Some additional topics include state restoration, targeting specific scenes and updating in response to push notifications.

Here are some resources for further study:

There are several WWDC videos on this topic. Two good ones from WWDC19 are: Introducing Multiple Windows on iPad and Architecting Your App for Multiple Windows.

Make sure you review Apple's documentation for UIScene, UIWindowScene, UISceneDelegate, and UIWindowSceneDelegate.

You may also want to checkout Apple's sample project: Supporting Multiple Windows on iPad.

If you have any questions or comments, join the discussion in the forum below.

Average Rating

5/5

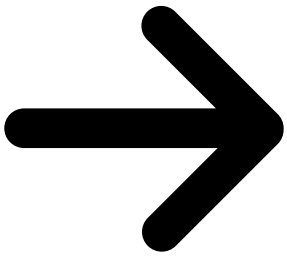
Add a rating for this content

Sign in to add a rating

5 ratings

raywenderlich.com Subscription

Full access to the largest collection of Swift and iOS development tutorials anywhere!



Learn more