CS275
Fall 2020

iOS: UINavigationController

BNRG CHAPTER 14

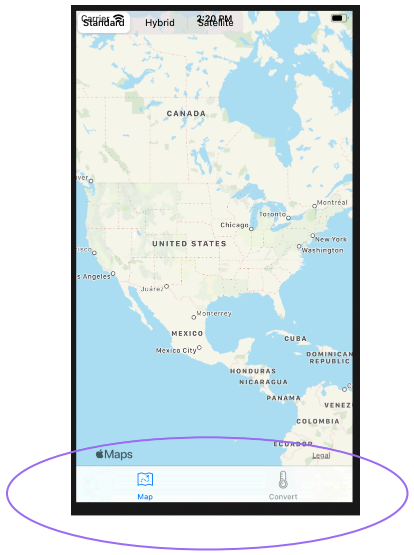
Topics

- Navigating between view controllers, using a navigation controller
- Responding to events, event handling, and dismissing the keyboard

Navigation

In an interface that presents a single screen (such as a phone), we need to be able to move from screen to screen

- one way is by using a tab-bar controller

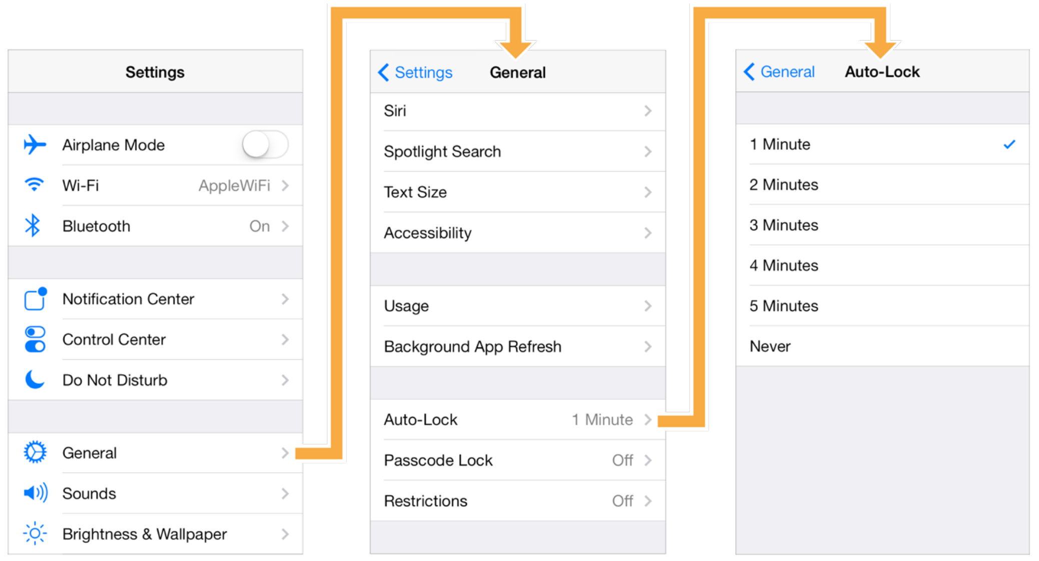


Drill-Down Interface

Another natural way to move from screen to screen is with a drill-down interface

- Device Settings is an example of a drill-down interface

A `UINavigationController` enables this navigation through different screens



Settings

Airplane Mode
Wi-Fi AppleWiFi
Bluetooth On
Notification Center
Control Center
Do Not Disturb
General
Sounds
Brightness & Wallpaper

Settings General

Siri
Spotlight Search
Text Size
Accessibility
Usage
Background App Refresh
Auto-Lock 1 Minute
Passcode Lock Off
Restrictions Off

General Auto-Lock

1 Minute ✓
2 Minutes
3 Minutes
4 Minutes
5 Minutes
Never

`UINavigationController`

A `UINavigationController` maintains an array of view controllers that have activated each other in a a linear sequence

It represents the list of view controllers as a stack

UINavigationController

Conceptually, the `UINavigationController` maintains a stack of view controllers

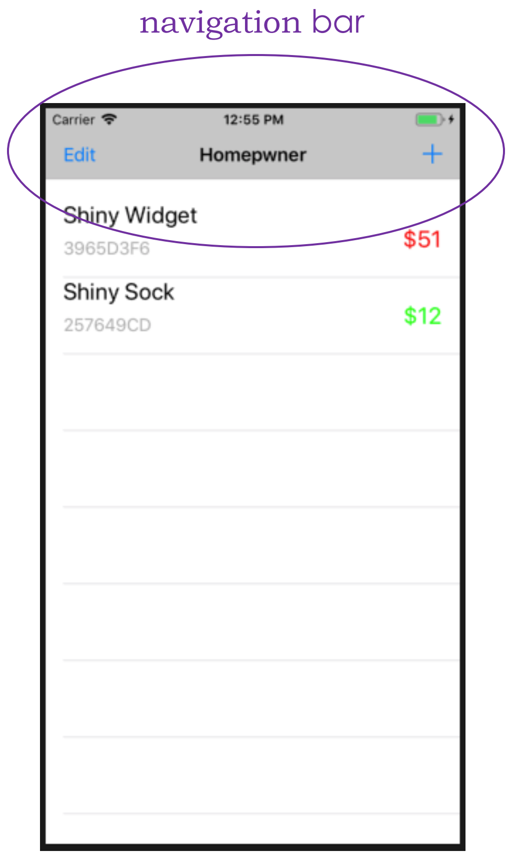
- When we give a new view controller to the `UINavigationController`, it pushes that new VC onto the stack, and that VC slides onscreen
- When we do "back" on the top VC, that VC is popped the stack, and the VC underneath appears

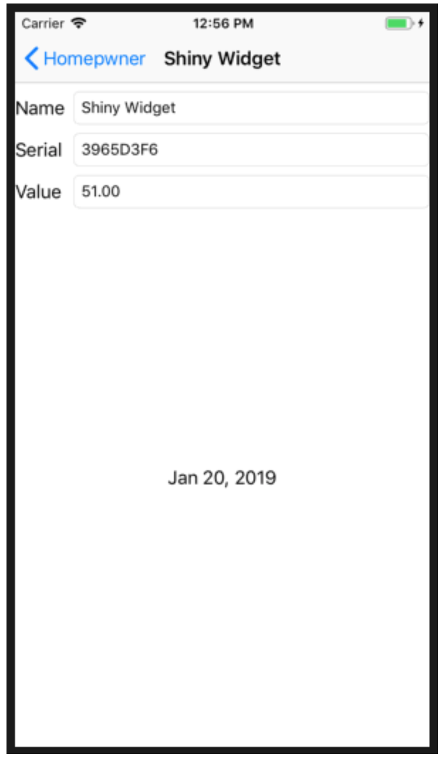
UINavigationController

The "root" VC is the VC at the bottom of the stack

The UINavigationController has two subviews

- a UINavigationBar, which enables navigation
- and the view of the topmost view controller





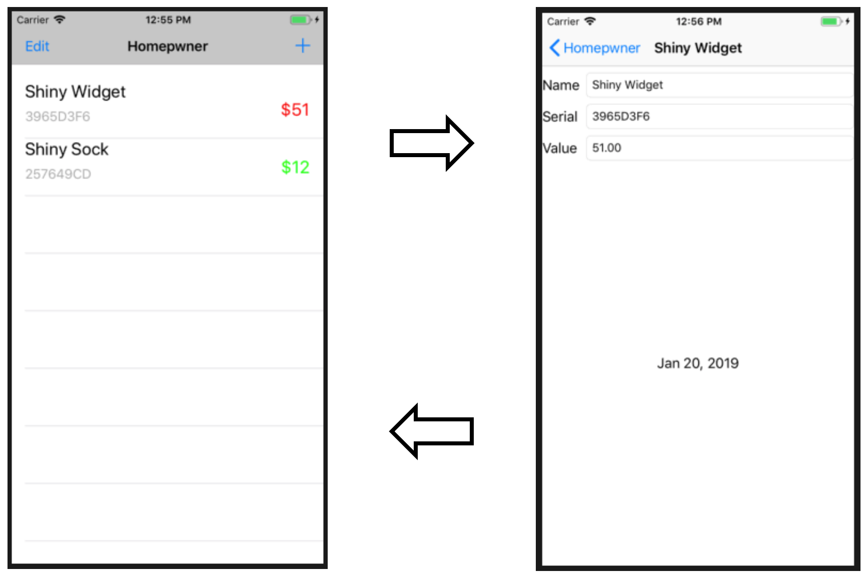
Name	Shiny Widget
Serial	3965D3F6
Value	51.00

Jan 20, 2019

Overview-Detail Interface

For an Overview-Detail interface, `UINavigationController` is a natural interface

- here, to enable navigation between the `ItermsViewController` and the `DetailViewController`

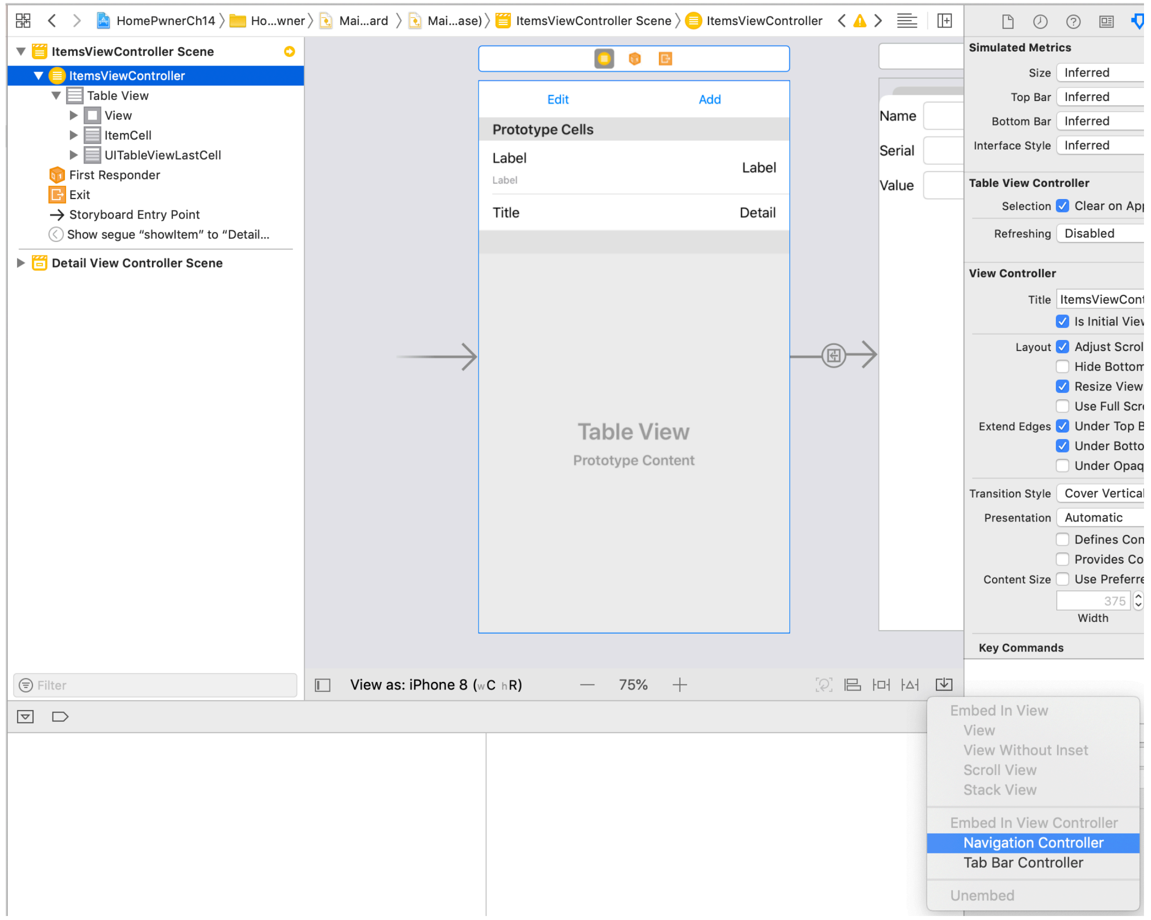


Shiny Widget
3965D3F6
$51
Shiny Sock
257649CD
$12

Name Shiny Widget
Serial 3965D3F6
Value 51.00
Jan 20, 2019

Creating a Navigation Controller

- To create a navigation controller, select a VC on the storyboard
- Then "embed in" a Navigation Controller



Simulated Metrics
Size: Inferred
Top Bar: Inferred
Bottom Bar: Inferred
Interface Style: Inferred

Table View Controller
Selection: ✓ Clear on App
Refreshing: Disabled

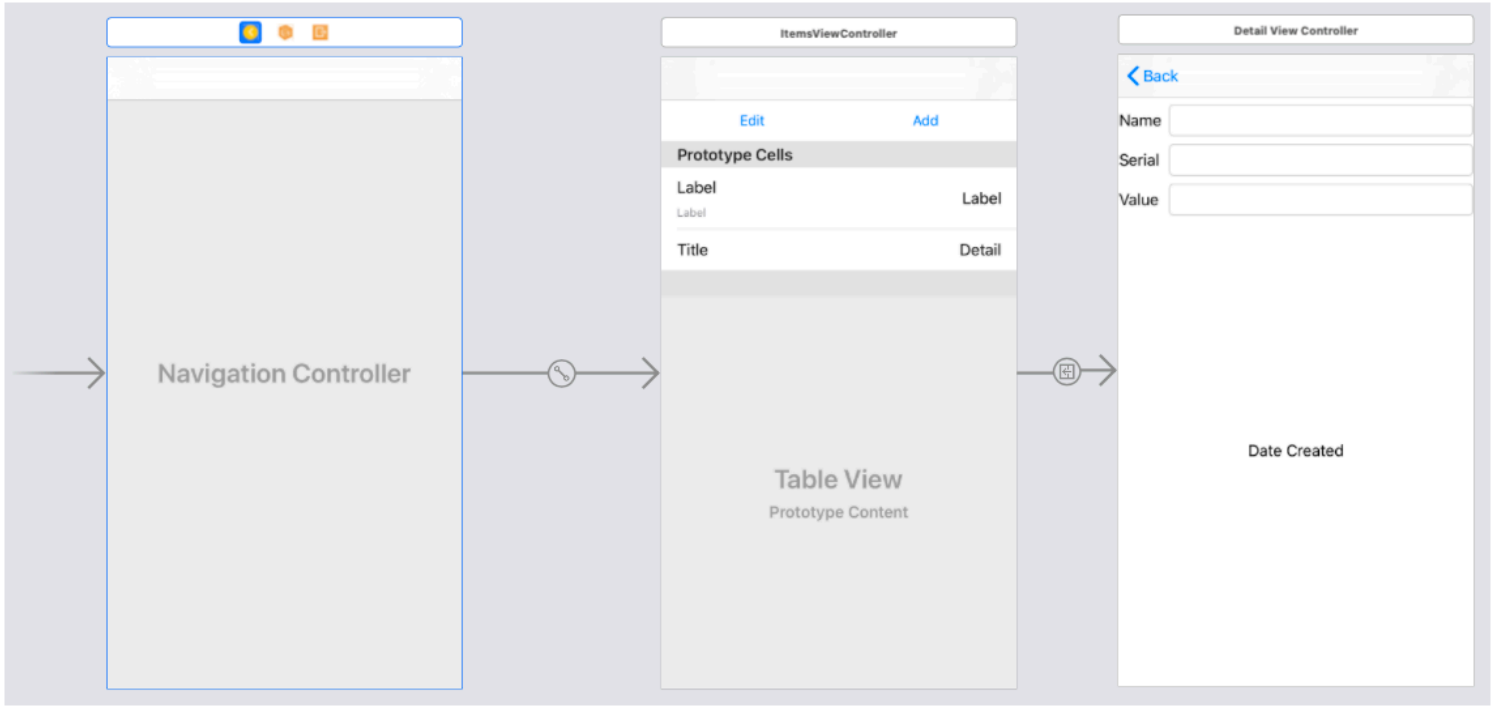
View Controller
Title: ItemsViewCont
✓ Is Initial View
Layout: ✓ Adjust Scrol
☐ Hide Bottom
✓ Resize View
☐ Use Full Scr
Extend Edges: ✓ Under Top B
✓ Under Botto
☐ Under Opaq
Transition Style: Cover Vertical
Presentation: Automatic
☐ Defines Con
☐ Provides Co
Content Size: ☐ Use Preferre
375
Width

Key Commands

Updated Storyboard

Here's what the storyboard will look like

- the navigation controller is now the entry point for the app



Initial View Controller

In app, one view controller is designated as the initial view controller

- this is the VC that will appear when the app starts

When we are using a navigation controller, it's the navigation controller itself that is the initial view controller

In order to initialize the app, we need to get access to the first VC that will appear

- this corresponds to the VC at the top of the navigation controller's stack

App Start-Up

And to set things up correctly in the app, we have to be able to perform actions during app start-up

Starting in iOS 13, iOS uses the concept of a scene to manage the presentation of an app's screens

AppDelegate and SceneDelegate

The AppDelegate manages the application lifecycle and setup

- independent of what's actually presented in the UI

The SceneDelegate is responsible for what is shown on the screen and manages the way that the UI of an app is presented, in a Scene Session

Apple split this up to support multi-window apps

- in an iPad
- and to support the case where two instances of an app are running on an iPhone

SceneDelegate Methods

Important callbacks in SceneDelegate.swift:

```
scene(_:willConnectTo:options:)
```

This is the first method called in the UISceneSession lifestyle

- creates the UIWindow
- sets the root VC

SceneDelegate Methods

`sceneWillResignActive(_:)`

`sceneDidEnterBackground(_:)`

These will be called if your window moves from the foreground

```
func sceneWillResignActive(_ scene: UIScene) {
    // Called when the scene will move from an active state to an inactive state.
    // This may occur due to temporary interruptions (ex. an incoming phone call).
}

func sceneDidEnterBackground(_ scene: UIScene) {
    // Called as the scene transitions from the foreground to the background.
    // Use this method to save data, release shared resources, and store enough scene-specific state information
    // to restore the scene back to its current state.
}
```

SceneDelegate Methods

SceneDidDisconnect(_:)

The scene has been disconnected from its session

```
func sceneDidDisconnect(_ scene: UIScene) {
    // Called as the scene is being released by the system.
    // This occurs shortly after the scene enters the background,
    // or when its session is discarded.
    // Release any resources associated with this scene that can be re-created
    // the next time the scene connects.
    // The scene may re-connect later, as its session was not neccessarily discarded
    // (see `application:didDiscardSceneSessions` instead).
}
```

SceneDelegate

Apple has actually made scene and window management quite complex

- in particular, with user state restoration

Here's a good blog post

- https://www.donnywals.com/understanding-the-ios-13-scene-delegate/

Getting the Initial View Controller

Before iOS13, as shown in the book

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession,
           options connectionOptions: UIScene.ConnectionOptions) {
    // Use this method to optionally configure and attach the UIWindow `window`
    //to the provided UIWindowScene `scene`.
    // If using a storyboard, the `window` property will automatically be initialized and attached to the scene.
    // This delegate does not imply the connecting scene or session are new (see
    // application:configurationForConnectingSceneSession` instead).
    guard let _ = (scene as? UIWindowScene) else { return }

    // create an ItemStore
    let itemStore = ItemStore()
    // access the ItemsViewController and set its item store
    let itemsController = window!.rootViewController as! ItemsViewController
    itemsController.itemStore = itemStore
}
```

here, we are assuming that `ItemsViewController` will be the initial view controller; but after embedding in a navigation controller, it won't be

With SceneDelegate, in iOS 13

Get a reference to the initial VC in `scene(_:willConnectTo:options:)`

```
scene(_:willConnectTo:options:) {
    guard let _ = (scene as? UIWindowScene) else { return }

    // create an ItemStore
    let itemStore = ItemStore()

    let navController = window!.rootViewController as! UINavigationController
    let itemsController = navController.topViewController as! ItemsViewController
    itemsController.itemStore = itemStore
}
```

A Better Way

Set the ItemStore in the AppDelegate's `application(_:didFinishLaunchingWithOptions:)`

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var itemStore: ItemStore!

    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions:
                              [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.

        itemStore = ItemStore()
        return true
    }
```

A Better Way

And then in SceneDelegate:

```
func scene(_ scene: UIScene, willConnectTo session: UISceneSession,
           options connectionOptions: UIScene.ConnectionOptions) {
    // Use this method to optionally configure and attach the UIWindow `window`
    // to the provided UIWindowScene `scene`.
    // If using a storyboard, the `window` property will automatically be initialized
    // and attached to the scene.
    // This delegate does not imply the connecting scene or session are new (see
    // `application:configurationForConnectingSceneSession` instead).
    guard let _ = (scene as? UIWindowScene) else { return }

    let appDelegate = UIApplication.shared.delegate as! AppDelegate
    let itemStore = appDelegate.itemStore

    let navController = window!.rootViewController as! UINavigationController
    let itemsController = navController.topViewController as! ItemsViewController
    itemsController.itemStore = itemStore
}
```

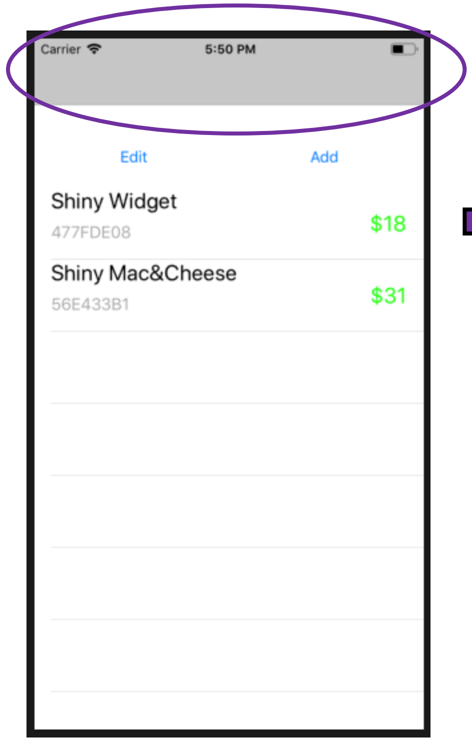
Navigation Bar

We defined a segue from `ItemsViewController` to `DetailViewController`

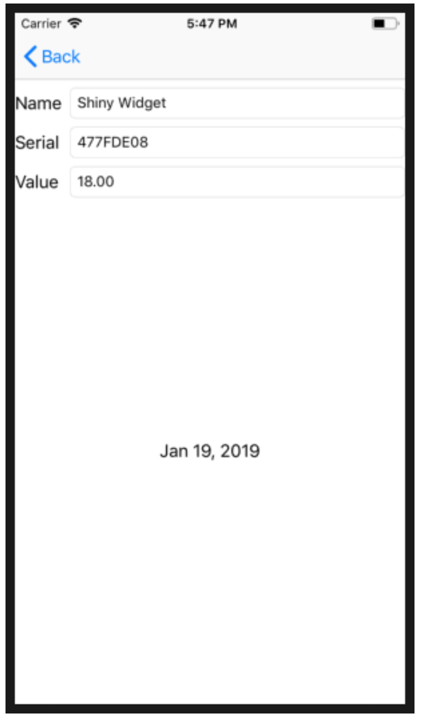
- the UINavigationController takes over and manages that segue
- in other words, it treats the segue from the table view to the `DetailViewController` as a navigation action

This is good

- "Embed in" really does mean embed in
  - the content size is adjusted to fit underneath the navigation bar



Shiny Widget 477FDE08 $18
Shiny Mac&Cheese 56E433B1 $31



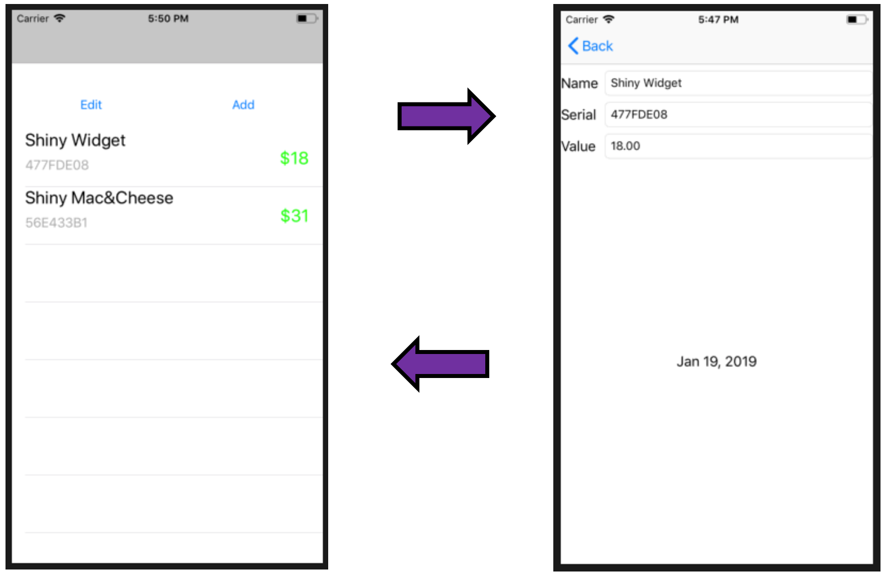
Name Shiny Widget
Serial 477FDE08
Value 18.00
Jan 19, 2019

But we must also define the content of the navigation bar on the root VC

Overview-Detail Interface

The natural interface for this kind of app is that we click on an item in the list (the overview view) and see the details for that item in the detail view

- then, if we make an edit to a field in the detail view, the underlying model is updated
- And the overview reflects that change
- This won't happen yet here
- To enable this to happen, we have to be notified when a navigation event is going to occur



Notifications from `UINavigationController`

When a `UINavigationController` is about to swap views, it calls two methods:

1. `viewWillDisappear(_:)` on the `UIViewController` that is about to be popped off the stack
2. `viewWillAppear(_:)` on the `UIViewController` that will then be on top of the stack

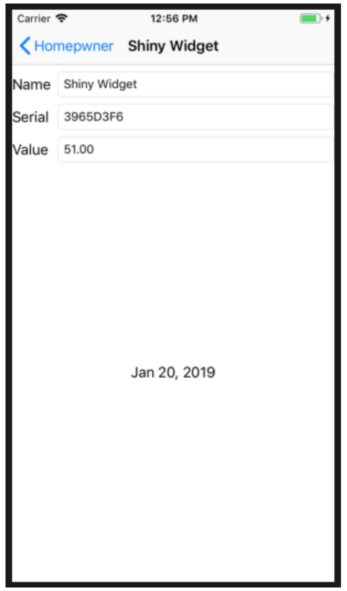
These methods give us a chance to update the model during a navigation action

- to reflect changes that have been made in the view

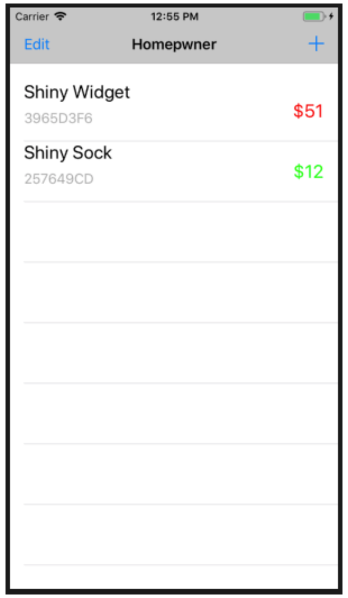
In each case, first call the superclass's implementation of the method

Updating an Item

If we make an edit to a field in `DetailViewController`, we want that change to be reflected in the model (in the Item itself)



Name Shiny Widget
Serial 3965D3F6
Value 51.00
Jan 20, 2019



Shiny Widget 3965D3F6 $51
Shiny Sock 257649CD $12

make a change here → and it should be reflected here

Updating an Item

If we make an edit to a field in `DetailViewController`, we want that change to be reflected in the model (in the Item itself)

- to do this, implement `viewWillDisappear(_:)` in `DetailViewController`
- and implement `viewWillAppear(_:)` in `ItemsViewController` to reload the items in the view

```
// in DetailViewController
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
    // "save" changes to item
    item.name = nameField.text ?? ""
    item.serialNumber = serialNumberField.text
    if let valueText = valueField.text,
        let value = numberFormatter.number(from: valueText) {
        item.valueInDollars = value.intValue
    } else {
        item.valueInDollars = 0
    }
}
```

```
// in ItemsViewController
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    tableView.reloadData()
}
```

Responding to Events

When I touch a UIButton, that button receives a touch event

- this is pretty obvious and is dictated by the raw geometry of the screen
- when I touch inside the effective coordinates of the button, the UI framework notifies the button

But some actions are not explicitly tied by geometric location to a specific UI view

- for example: a shake action

The question is: which UI component should receive notification of the action?

- the on-screen keyboard is similar
- when I type in the on-screen keyboard, which text field should receive the text that I type?

The Keyboard

When the user touches inside a text field. the device keyboard appears

- in the Simulator, "Hardware -> Toggle Software Keyboard"
- this behavior is built into `UITextField`

Sometimes it's necessary to control the default behavior of the keyboard

- for example, the $\boxed{\text{return}}$ key to dismiss the keyboard

Event Handling

When the user touches on the screen, Cocoa Touch creates an event

- in this case, a touch event

The event is delivered to the appropriate view

- for example, a button or a text field (depending on where the touch occurred)
- the view can choose to ignore the event

Sometimes it's not clear which view should receive the notification

- for example, a shake event
- or text entry on the keyboard

First Responder

First responder status is a mechanism whereby a UI element can signal to the UI framework that it wants to receive notification of an event

- there can be only one first responder at a time
- it's like a flag that is passed among views

First Responder

`UITextField` and `UITextView` have the use of first-responder status built in:

- when you touch a `UITextField` or `UITextView`, it triggers the system to put the keyboard on screen
- and it then becomes the first responder for the text that's typed

glossary

- `UILabel`: The `UILabel` class implements a read-only text view.
- `UITextField`: A `UITextField` object is a control that displays editable text and sends an action message to a target object when the user presses the return button.
- `UITextView`: The `UITextView` class implements the behavior for a scrollable, multiline text region.

Dismissing the Keyboard

The keyboard will disappear when the the view that is the first responder gives up its first-responder status

Desired behavior: have the keyboard disappear when the Return key is pressed

- in other words, have the text field surrender first-responder status when the Return key is pressed

Implement `textFieldShouldReturn(_:)` in `DetailViewController`

- this method is called whenever the Return key is pressed
- and in your implementation for this method, call `resignFirstResponder()` on the text field that is currently getting text

Listening for the Return Key

When the return key is pressed

- the method `textFieldShouldReturn(_:)` in `UITextFieldDelegate` is called

So to listen for the return key:

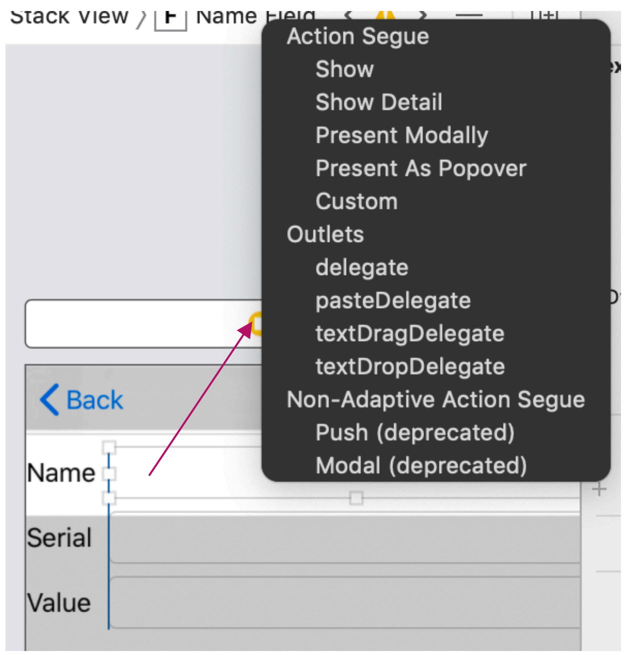
- have the view controller conform to `UITextFieldDelegate` and implement `textFieldShouldReturn(_:)`

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

    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        textField.resignFirstResponder()
        return true
    }
}
```

Listening for the Return Key

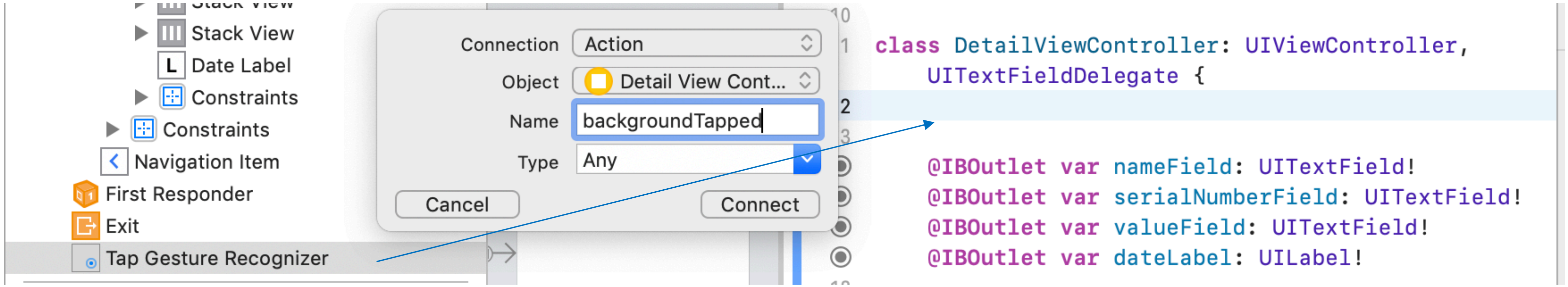
Also need to connect the delegate property of each text field to the `DetailViewController`

- control-drag from each text field to the `DetailViewController` and select delegate on the pop-up



Dismissing by Tapping Elsewhere

- Another nice UI behavior: dismiss the keyboard when the user taps anywhere else on the `DetailViewController`
- To do this, add a `TapGestureRecognizer` to the `DetailViewController`
- Control-drag from the tap-gesture recognizer to the implementation (the code) of `DetailViewController`

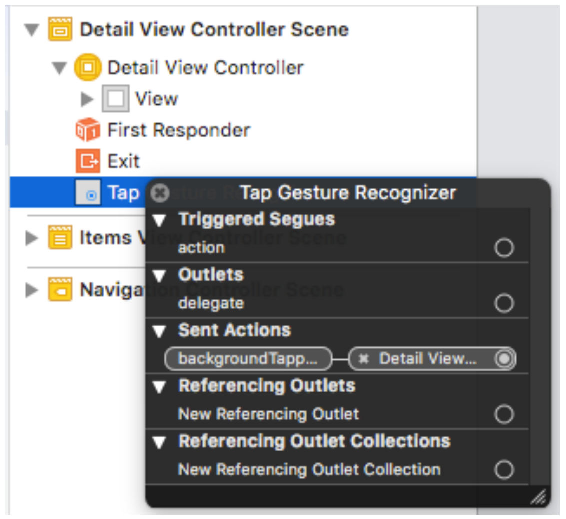


Connection: Action
Object: Detail View Cont...
Name: backgroundTapped
Type: Any

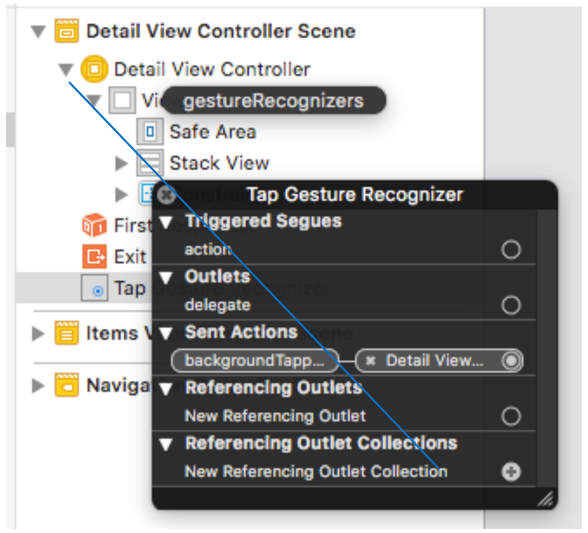
```
class DetailViewController: UIViewController,
    UITextFieldDelegate {


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

Gesture Recognizer

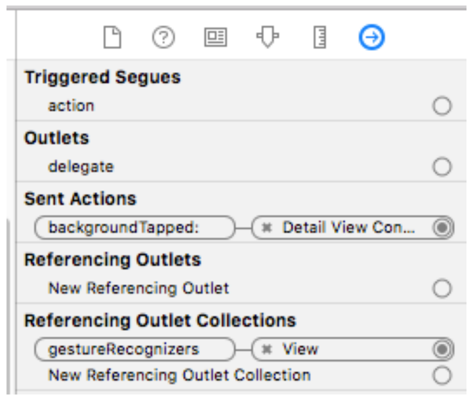
Make sure the "Referencing Outlet Collections" shows the View as the outlet for the gestureRecognizers; if it doesn't do the following



1. right-click on the Tap Gesture Recognizer



2. click on the circle in "New Referencing Outlet Collection" drag to the View; then click on gestureRecognizers



Here's what you should see for the outlets of the Tap Gesture Recognizer

And a Little Polishing

When the user taps "Back" in the navigation bar, the keyboard will disappear instantly (if it's visible), without any nice animation

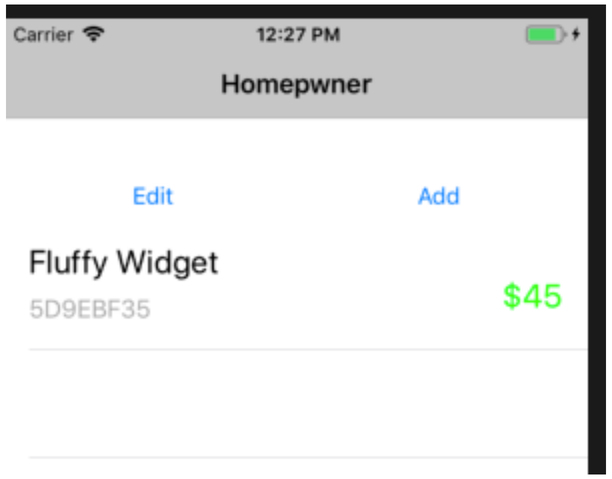
Polish: intercept the `viewWillDisappear(_:)` in `DetailViewController` and call `endEditing(_:)`

Since there is already an implementation of `viewWillDisappear(_:)` in `DetailViewController`, just add the `view.endEditing(true)` call to it

Modifying the Navigation Bar

Every `UIViewController` has a property `navigationItem` of type `UINavigationItem`

- this property tells the `UINavigationBar` what to display when that view controller's view is at the top of the stack
- at the simplest, it's a title ("Settings", "Display", etc.)
- by default, it's empty (which is why the navigation bar we created is empty)
- in Main.storyboard, give the ItemsViewController a name

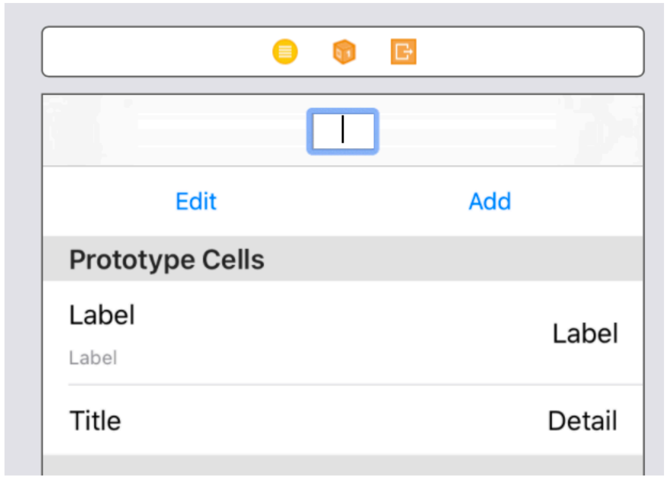


Modifying the Navigation Bar

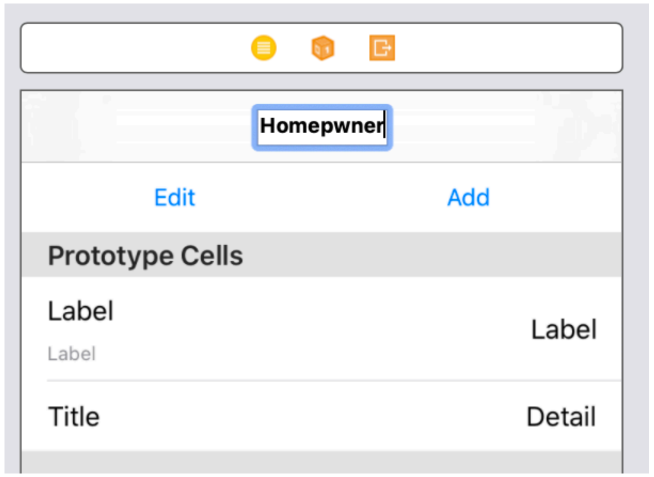
Simplest thing: a text string for the navigation bar

If the title for a view controller will not change as the app runs, then we can set the title in the storyboard

- by double-clicking in the center of the navigation bar



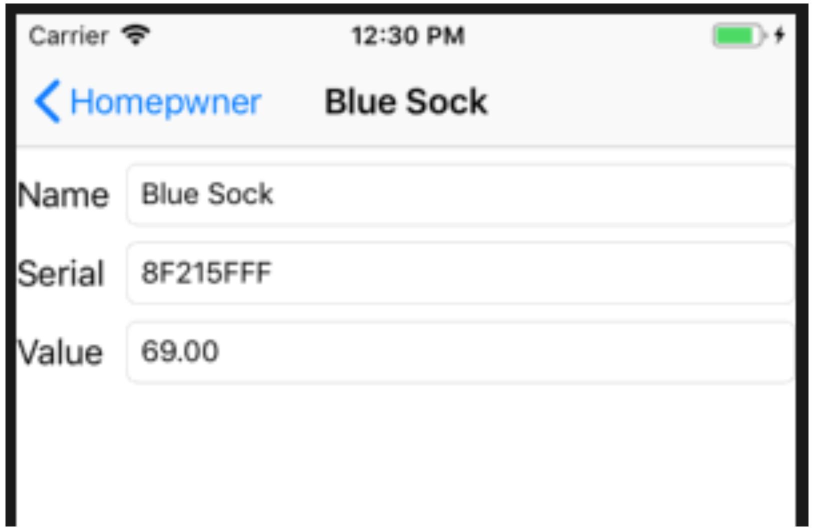
⇨



Property Observer

- For the detail view, the navigation bar should display the name of the item
- But, the name is not fixed—depending on which item is being shown, the title will change
- Solution: have the code inform us when the Item property in the `DetailViewController` changes
- That way, we can change the title in the navigation bar
- To do this, we can put a property observer on the Item in the `DetailViewController`
  - so when we change the name of the Item, we can change the title in the navigation bar

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


Carrier 12:30 PM

Homepwner Blue Sock

Name Blue Sock

Serial 8F215FFF

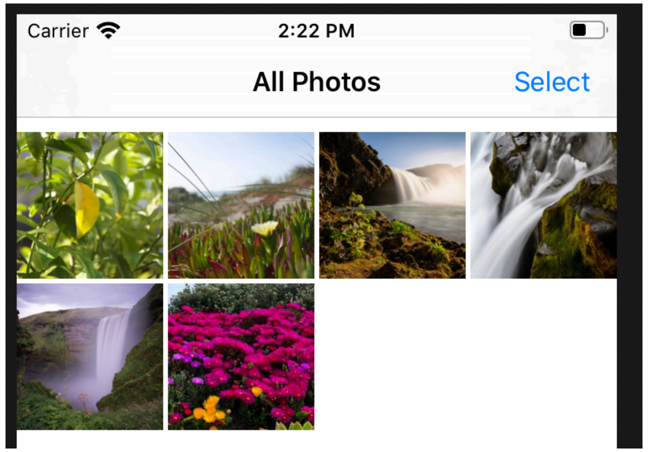
Value 69.00

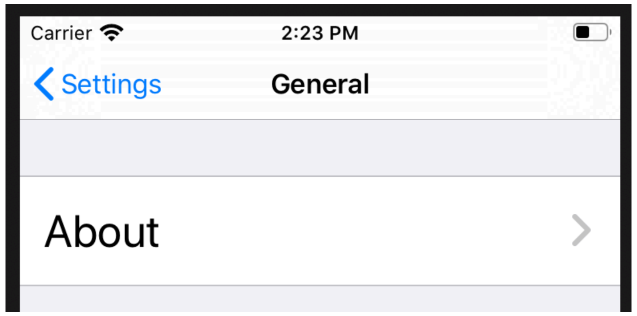
Customizing the Navigation Bar

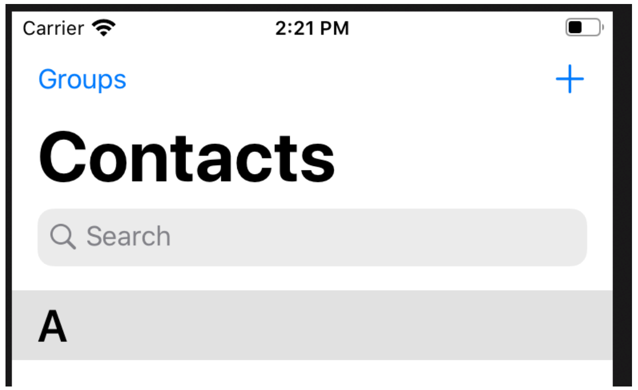
The navigation bar can have a subclass of UIView sit its middle

- Or it can have a title string in the middle

In addition, the navigation bar also has space for a button on the left and a button on right



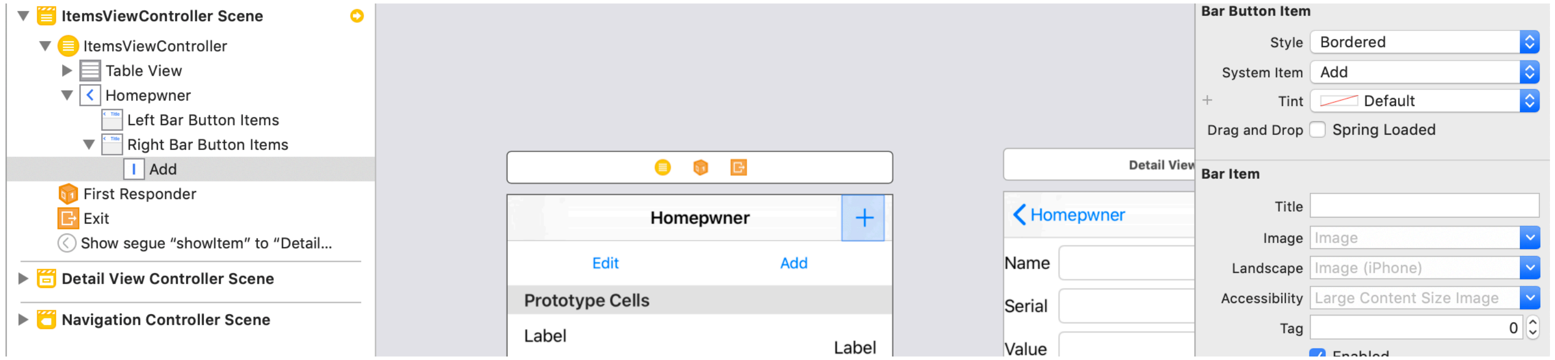




Configuring the Navigation Bar

To add a "+" button to the ItemsViewController's navigation bar

1. add this to Main.storyboard: by dragging a Bar Button Item to the navigation bar of the Items View Controller
2. change the System Item to "Add" will change its icon to a +
3. must set its action as `addNewItem`
4. and the existing `addNewItem(_ sender: UIButton)` will have to change to `addNewItem(_ sender: UIBarButtonItem)`



Bar Button Item

Style: Bordered
System Item: Add
Tint: Default
Drag and Drop: Spring Loaded

Bar Item

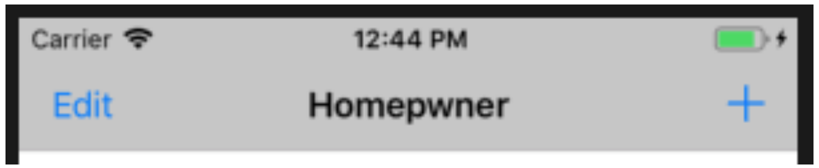
Title:
Image: Image
Landscape: Image (iPhone)
Accessibility: Large Content Size Image
Tag: 0
Enabled

Configuring the Navigation Bar

Add an "Edit" button to the ItemsViewController's navigation bar

- view controllers expose a bar button item that will automatically toggle their editing mode
- but it's not possible to access this button through Interface Builder—must add the button programmatically
- have to add this programmatically, in the initializer of the view controller or in `viewWillAppear(_:)`

This edit button toggles editing mode automatically (it's built in)


Carrier
12:44 PM
Edit
Homepwner
+