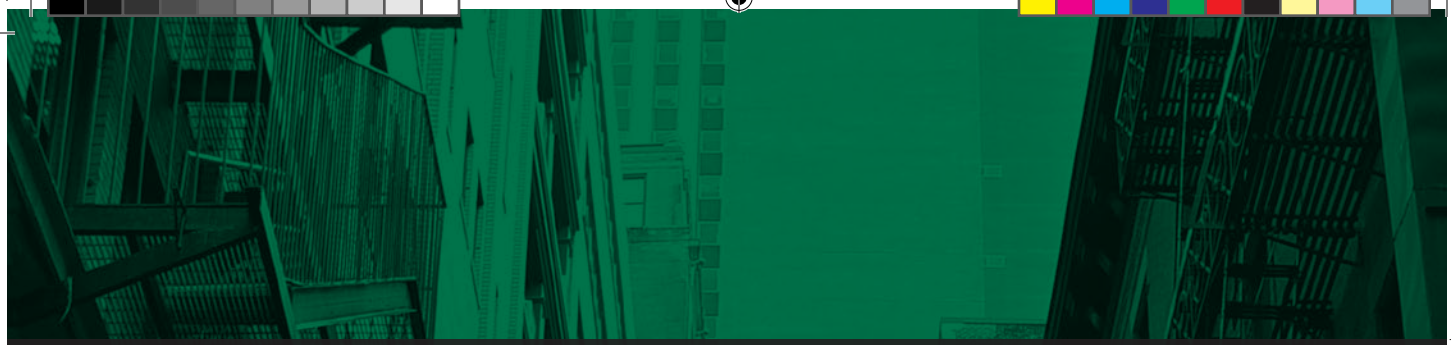
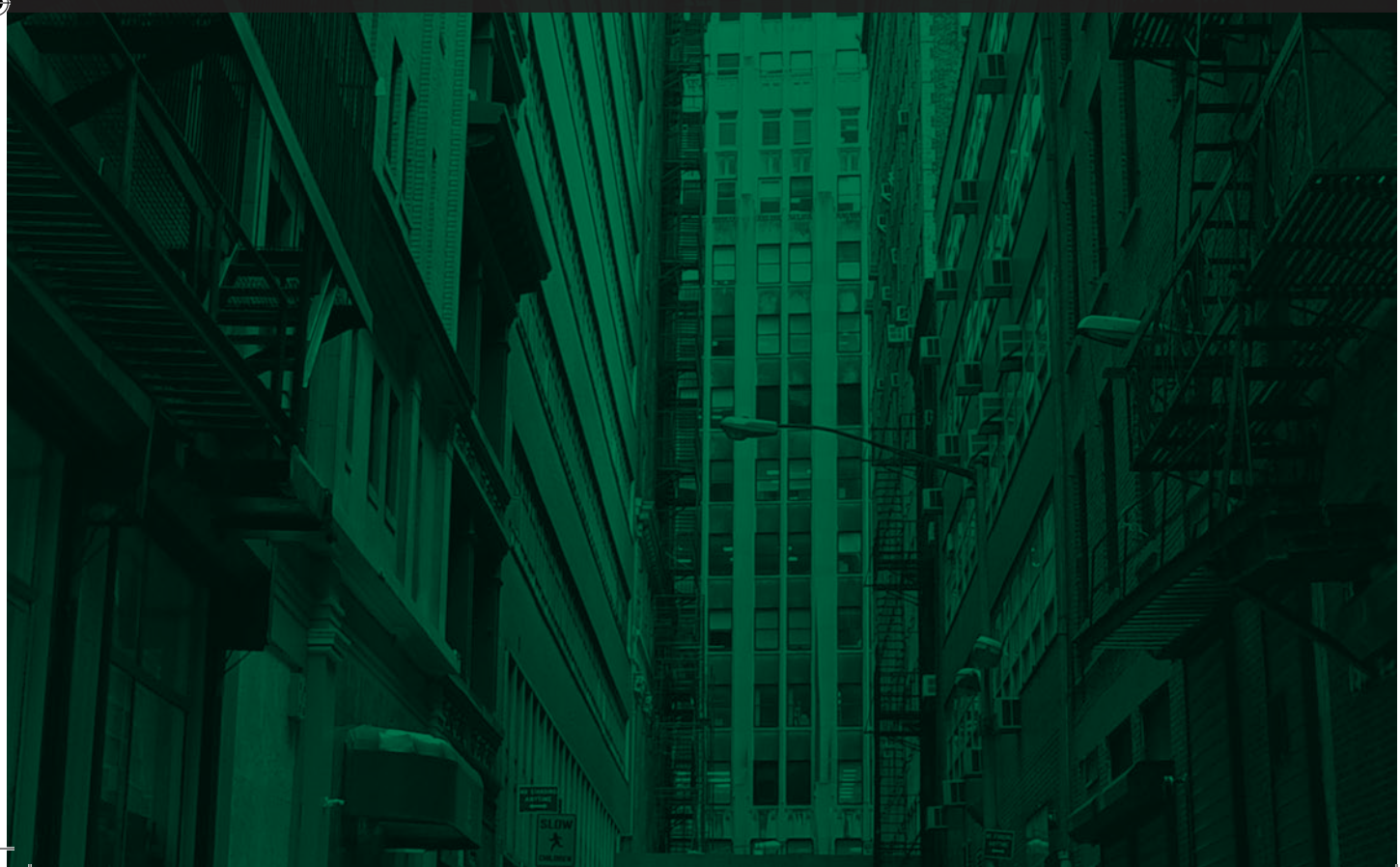


BONUS

# UI State Preservation and Restoration



iOS apps try to maintain the illusion that they are running in the background. When the user presses the home button, the application disappears. The system freezes the app's current state in memory. Then, when the user returns to the app, it restores that state exactly where the user left off.

The problem is, if the system needs additional memory while our application is in the background, it will silently kill our application and recycle its memory.

Every time we go to the background, we must assume our app could be killed. This means even quick tasks like checking email or answering the phone could result in our application being relaunched and the user losing their current position and unsaved work.

As much as possible, we want to make the difference between returning from the background and launching as transparent as possible. In an ideal universe, the user should not be able to tell the difference. They can leave the app, do whatever they want, and return with the faith that their work will remain exactly as they left it.

# SAVING THE USER INTERFACE

The theory is simple, actually pulling it off is not a trivial task, and the real world often requires a few compromises. Fortunately, UIKit offers some support in the form of State Preservation and Restoration. This is an opt-in feature, but once we enable it, the system will automatically record the current state of our application whenever we go to the background. Then, the next time our application launches, it will restore that state.

To preserve our state, we need to do the following steps:

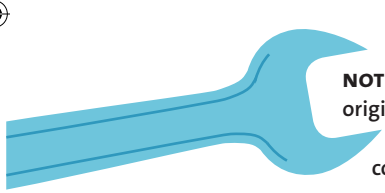
1. Tell the system to preserve our state.
2. Tell the system which view controllers to preserve.
3. Save any relevant data for those view controllers during the encoding step.

To restore, we do the following:

1. Tell the system to restore our state.
2. Tell the system which view controllers to restore.
3. Load any relevant data for those view controllers during the decoding step.

Fortunately, the preserving and restoring steps largely parallel each other, so we can look at both sides of the operation at the same time.

## ENABLE STATE PRESERVATION AND RESTORATION



**NOTE: The code examples used throughout the rest of this chapter was originally planned for the end of Chapter 5 of Creating iOS Apps: Develop and Design Second Edition. However, you should be able to add this code to your Health Beat project any time after Chapter 5.**

Open `AppDelegate.m`. At the bottom of the `@implementation` block, add the following two methods:

```
- (BOOL)application:(UIApplication *)application
shouldSaveApplicationState:(NSCoder *)coder
{
    return YES;
}

- (BOOL)application:(UIApplication *)application
shouldRestoreApplicationState:(NSCoder *)coder
{
    return YES;
}
```

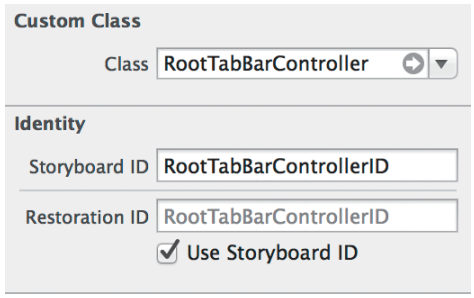



**FIGURE 1** Setting the restoration ID

Here, we just tell the system that we want to both preserve and restore our UI state.

While just returning `YES` is a very common pattern, we can also examine our coder and dynamically return a result. For example, we might want to check when the UI's state was last saved. If it was a significant time ago, we may want to just relaunch the app from the beginning. We can access the timestamp in `application:shouldRestoreApplicationState:` as shown here:

```
NSDate *timeStamp = [coder
decodeObjectForKey:UIApplicationStateRestorationTimestampKey];
```

## SELECT THE VIEWS AND VIEW CONTROLLERS TO RESTORE

For the step 2s, we need to provide a restoration ID for each of the controllers and views we want to preserve. Each object must have a unique restoration path—this is the series of restoration identifiers, starting with the root view controller and walking down the view controller hierarchy until it reaches the desired object. Objects are loaded and saved based on their restoration path.

We can easily set the restoration ID in our storyboard. Open `Main.storyboard`. Select the Root Tab Bar Controller, and switch to the Identity inspector. We want to set Storyboard ID to **RootTabBarControllerID** and then click the Use Storyboard ID checkbox (**Figure 1**).

The Storyboard ID is the ID we can use to programmatically load a view controller from a storyboard. It's always best to always use the same ID for both the storyboard and restoration.

By setting the controller's restoration ID, we are telling it to both save and load its state during UI State Preservation and Restoration. Specifically, the tab bar controller will record which tab is currently selected.

Next, select the History Navigation Controller and set its Storyboard ID and Restoration ID to **HistoryNavigationControllerID**. This will tell it to save and restore all the view controllers in its stack.

Repeat this for the History Table View Controller. This time use **HistoryTableViewControllerID**. While this lets us save and restore the table view controller, it won't save and restore the table's state. So, we need to select the table view as well and give it its own Restoration ID. If you've deciphered my clever naming scheme, you can probably guess that it should be **HistoryTableViewID**.

There are a couple of important points here. First, the table view controller only needs an ID to pass the baton along, allowing the automatic loading and saving of its table view. If it did not have an ID, the system would not save and restore its state. It also would not save and restore its view hierarchy or any of its child view controllers—even if they have their own restoration IDs.

Second, by default the table view will record its scroll position and the index path for the currently selected row. This works for our application, since the content of the table cannot change while the application is inactive. Unfortunately, as soon as we enable iCloud syncing, things get more complicated. As a result, we will need a somewhat more complex procedure to record our table's state. We will look at this in Chapter 6, "iCloud Syncing."

Finally, notice that views do not have Storyboard IDs. They have only Restoration IDs. This is because we cannot load views directly from storyboards—only view controllers.

Repeat the procedure for our Graph View Controller (**GraphViewControllerID**) and Entry Detail View Controller (**EntryDetailViewControllerID**). These controllers won't save any state, but if we don't give them an ID, the system will assume we don't want to restore to their position and will roll back to the next logical point.

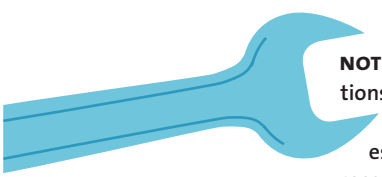
For the graph view controller, this means instead of restarting with the graph tab selected, it will default to the starting tab (our history tab). For the Entry Detail Scene, it means the navigator will simply show our history table instead.

Finally, we do not want to add an ID for our Add Entry View Controller. State restoration and custom modal transitions do not play well together. The simplest solution is to just ignore the Add Entry scene when preserving and restoring our UI state.

Run the application. Navigate to a view, and then press the home button to put it in the background. Kill the application from Xcode and relaunch it. It should return—more or less—to where we left off.

When the application is relaunched, it will show a snapshot of the UI from our previous launch. This will then cross fade to our restored UI. However, we have to be a little careful with the timing. State restoration finishes before our asynchronous document loading. This means we cannot rely on our document being in place as the state is restored. In some ways, this works to our advantage, since the graph and history scenes will be automatically updated, but it means we may temporarily transition to an empty table or graph before the data is ready. We will also have to put in a little extra work to get the Entry Detail scene to work correctly.

Currently, the detail view shows the previous snapshot and then fades to all zeros. To fix this, we need to pass the correct data to the detail view when our document loads. But, to do this, we need to save and load some additional data.



**NOTE:** If you double-click the home button to bring up the list of applications and then up-swipe on an application to force close it, it will also delete the state preservation data for that app. This acts as an emergency escape hatch for the user, letting them clear the preserved data, just in case their application happens to get into a bad state during restoration. This also lets us clear our state data during testing.

### RESTORATION CLASS

Along with the restoration ID, we may also need to provide a restoration class—this class adopts the `UIViewControllerRestoration` protocol and then must implement the `+viewControllerWithRestorationIdentifierPath:coder:` method. In this method, we can create and configure the view or view controller that is being restored.

If we don't provide a restoration class, the system will check to see whether it has already loaded an object with the matching restoration path. If it has, it simply returns that object. If it hasn't, it will attempt to load the object from the storyboard.

We should never use a restoration class for any objects that the system automatically creates for us when the application loads. Otherwise, we may end up with two copies of these objects. In Health Beat this includes our `RootTabBarController`, `HistoryNavigationController`, `HistoryTableViewController`, and `GraphViewController`, as well as each controller's view hierarchy.

Additionally, we don't need to provide a restoration class, if the object can simply be loaded from the storyboard. Again, in Health Beat, we aren't going to do any special configuration or processing of the `EntryDetailViewController`, so we can just let the system automatically load it from the storyboard. That means none of the controllers or views needs a restoration class.

## LOAD AND SAVE ADDITIONAL DATA

In the step 3s, we load and save any additional data and perform any last-minute configuration. There are three methods we can override to save, load, and update our data: `encodeRestorableStateWithCoder:`, `decodeRestorableStateWithCoder:` and `applicationFinishedRestoringState`.

The encode and decode methods are called during state preservation and state restoration, respectively. They each have a single argument, an `NSCoder` object that we can use to save or load our data.

The `applicationFinishedRestoringState` method is called only during state restoration, after all of the object decoding has finished. We can override this method to perform configuration tasks that depend on other objects in the archive. By the time this method is called, everything in the archive has been instantiated and is ready to use. Conceptually, this is similar to the way we use `viewDidLoad` to configure our views after everything has loaded from the nib file.

As we take a closer look at the encode and decode methods, they should look familiar. The State Preservation and Restoration use the same `NSCoding`/`NSCoder` pattern we used to save our document data. In this case, the coders are not `NSKeyedArchiver`/`NSKeyedUnarchiver` objects. They have the same interface, but the state preservation coders are specifically designed to set up the user interface.



For example, if we are saving a view or view controller, the state preservation coder looks to see whether it has already saved an object with a matching restoration path. If it has, it simply adds another reference to the matching saved object. If no match is found, it records the object's restoration path and then calls its `encodeRestorableStateWithCoder:` method.

When restoring, it checks to see whether it's already loaded an object with a matching restoration path. If it has, it simply returns a reference to that object. If not, it loads the correct object from the given restoration path and then calls `decodeRestorableStateWithCoder:` on that object.

Using these methods requires a bit of thought. We want to save enough data to restore our application to a functional state—but we don't want to just set up a fake façade. And we never want to use this to save our document's data.

Most of the time, if we wanted to record a piece of document data, we would record some sort of ID here, and then we would use that ID to request the actual data from our document when the system restored our controller.

In Health Beat, this causes some complications. Since we are asynchronously loading our document, the document's data won't be available until after the state restoration has finished. So, we need to store the information and use it to update the interface once the document data becomes available.

Let's start by making the `EntryDetailViewController`'s `updateUI` method public. Open `EnteryDetailViewController.h` and add the following method declaration in the `@interface` block:

```
- (void)updateUI;
```

We also want to reorganize the method definition in the `.m` file so that it's in a `#pragma mark - Public Methods` section, instead of the private methods section.

Next, we have another small problem to work around. Apparently, the table view's ability to save and restore its scroll position works only as long as our history table view controller is the top of the stack. Once we've selected a detail view, state preservation stops recording both the scroll position and the index path.

I suspect this happens because at the point where the index path is restored, the table view is empty. This means the index path is invalid. This seems to prevent the table view from restoring at all. Unfortunately, that means we must manually record and restore both the scroll position and the selected index path. And we need a place to stash that data until we're ready to use it.

We're going to store all the actions we need to update our table view in a block. This means we need a property to hold the block. I find that it's easiest to use blocks in properties and method arguments if I create a block type first.

Open HistoryTableViewController.m. At the top of the file, just under the #import lines, add the following:

```
typedef void (^CompletionBlock)(void);

static NSString * const HistoryTableViewControllerIsShowingDetailKey =
@"HistoryTableViewControllerIsShowingDetailKey";

static NSString * const HistoryTableViewControllerScrollOffsetKey =
@"HistoryTableViewControllerScrollOffsetKey";

static NSString * const HistoryTableViewControllerSelectedIndexPathKey =
@"HistoryTableViewControllerSelectedIndexPathKey";
```

This creates a block type named CompletionBlock that does not take any arguments and does not return any values. Then we define three static keys, which we use to encode and decode our data.

Now, in the class extension, add a property for our block.

```
@property (strong, nonatomic) CompletionBlock restorationCompletionBlock;
```

Next, we need to override the encodeRestorableStateWithCoder: method. Add this after the navigation methods, as shown here:

```
#pragma mark - State Restoration Methods

- (void)encodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super encodeRestorableStateWithCoder:coder];

    BOOL isDisplayingEntryDetailView =
    [self.navigationController.topViewController
     isKindOfClass:[EntryDetailViewController class]];

    [coder encodeBool:isDisplayingEntryDetailView
               forKey:HistoryTableViewControllerIsShowingDetailKey];

    if (isDisplayingEntryDetailView)
    {
        [coder
         encodeCGPoint:self.tableView.contentOffset
         forKey:HistoryTableViewControllerScrollOffsetKey];
```

```
        [coder
         encodeObject:[self.tableView indexPathForSelectedRow]
         forKey:HistoryTableViewControllerSelectedIndexPathKey];
    }
}
```

We start by calling `super`. Then we check to see whether our navigation controller is currently displaying our `EntryDetailViewController`. We store that value, and if it is `YES`, we also store the current scrolling offset and the currently selected index path.

Next, we need to implement the `decodeRestorableStateWithCoder:` method as shown here:

```
- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    [super decodeRestorableStateWithCoder:coder];

    BOOL isDisplayingEntryDetailView =
    [coder decodeBoolForKey:HistoryTableViewControllerIsShowingDetailKey];

    if (isDisplayingEntryDetailView)
    {

        CGPoint offset =
        [coder decodeCGPointForKey:
         HistoryTableViewControllerScrollOffsetKey];

        NSIndexPath *indexPath =
        [coder decodeObjectForKey:
         HistoryTableViewControllerSelectedIndexPathKey];

        __weak HistoryTableViewController *_self = self;
        self.restorationCompletionBlock = ^{
            [_self.tableView
             selectRowAtIndexPath:indexPath
             animated:NO
             scrollPosition:UITableViewScrollPositionNone];

            _self.tableView.contentOffset = offset;
        };
    }
}
```

Again, we start by calling super. Then we extract the Entry Detail scene's state from the archive. If we are displaying the detail scene, we extract our offset and our selected index path. Then we store a block that selects the correct row in the table and then update its scroll offset.

Now, we need to call and clear this block when our document loads. Navigate to the `weightHistoryDocument:stateDidChange:` method, and modify it as shown here:

```
- (void)weightHistoryDocument:(WeightHistoryDocument *)document
                stateDidChange:(UIDocumentState)state
{
    if ((state == UIDocumentStateNormal) ||
        (state == UIDocumentStateClosed))
    {
        [self.tableView reloadData];
    }

    if (state == UIDocumentStateNormal)
    {
        id topController = self.navigationController.topViewController;
        if ([topController isKindOfClass:
            [EntryDetailViewController class]])
        {
            if (self.restorationCompletionBlock)
            {
                self.restorationCompletionBlock();
                self.restorationCompletionBlock = nil;
            }
            [self configureWeightEntryViewController:topController];
        }
    }
    else
    {
        if ([self.presentedViewController
             isKindOfClass:[AddEntryViewController class]])
        {
            [self dismissViewControllerAnimated:YES completion:nil];
        }
    }

    [self updateAddButton];
}
```

When our document changes to the normal state (for example, because the document has successfully loaded), we check to see whether our navigation controller's top view controller is an Entry Detail View Controller. If it is, we check to see whether we have a restoration block. If we do, we call the block and clear it. Then we call our `configureWeightEntryViewController:` method to pass the correct data to our Entry Detail View Controller.

This gets the data to our Entry Detail scene, but it doesn't update the scene's UI. Navigate down to the `configureWeightEntryViewController:` method. Add the following line to the bottom of this method, right before the closing curly bracket:

```
[controller updateUI];
```

With this in place, our state restoration should work, even for the detail scene.

## WRAPPING UP

That's it. As you can see, adding state restoration is conceptually easy—but things rapidly get more complex once real code is involved. Fortunately, we don't have to implement a complete state restoration system all at once. We can start by simply capturing the low-hanging fruit, and then determine how far we want to push things. It may not be perfect in all cases, but we should be able to cover most situations with a minimal amount of effort.

## OTHER RESOURCES

- **iOS App Programming Guide: State Preservation and Restoration**

  *iOS Developer's Library*

  While the entire iOS App Programming Guide is worth reading, the section on State Preservation and Restoration provides detailed information about the State Preservation and Restoration process.