iOS 18: Notable UIKit Additions

// Written by Jordan Morgan
// Jun 3rd, 2024
// Read it in about 4 minutes
// RE: UIKit

This post is brought to you by Emerge Tools, the best way to build on mobile.

A.I. - amirite?

While this year's keynote was heavy on Apple Intelligence (that's what you thought A.I. stood for, right?) — our timeless user interface framework cracked on.

So, as is tradition - here are some notable UIKit additions in iOS 18. If you want to catch up on this series first, you can view the iOS 11, iOS 12, iOS 13, iOS 14, iOS 15, iOS 16 and iOS 17 versions of this article.

Automatic Trait Tracking

Oh **`registerForTraitChanges()`**, we hardly knew ye. In iOS 18, we get automatic trait change tracking — in *some* cases (so **`registerForTraitChanges()`** isn't entirely going away, but its primary use case is).

Consider this code:

```
class CustomBackgroundView: UIView {
    override init(frame: CGRect) {
        super.init(frame: frame)
        registerForTraitChanges([UITraitVerticalSizeClass.self], action: #
    }

    override func layoutSubviews() {
        super.layoutSubviews()

        if traitCollection.verticalSizeClass == .regular {
            backgroundColor = .blue
        } else {
            backgroundColor = .orange
        }
    }
}
```

We register to hear about **`verticalSizeClass`** changes, and run **`layoutSubviews()`** when it does. In iOS 18, it looks like this:

```
class CustomBackgroundView: UIView {
    override func layoutSubviews() {
        super.layoutSubviews()

        if traitCollection.verticalSizeClass == .regular {
            backgroundColor = .blue
        } else {
            backgroundColor = .orange
        }
    }
}
```

You can just...skip the register bit entirely! UIKit will note which traits we're interested in within the function, and invoke **`layoutSubviews()`** again when it changes. This only works in a few scenarios though, namely — "update" or invalidation type of lifecycle functions:

1. For views: **`layoutSubviews()`**, **`updateConstraints()`** and **`draw(CGRect)`**.
2. For view controllers: **`viewWillLayoutSubviews()`**, **`viewDidLayoutSubviews()`**, **`updateViewConstraints()`**, and **`updateContentUnavailableConfiguration(using:)`**.
3. In presentation controllers: **`containerViewWillLayoutSubviews()`** and **`containerViewDidLayoutSubviews()`**.
4. Inside buttons, table view headers and footers, and collection or table view cells (remember those!): **`updateConfiguration()`** and **`configurationUpdateHandler`**.
5. Collection view compositional layouts: **`UICollectionViewCompositionalLayoutSectionProvider`**.

That's a nice quality of life update — I'm sure Apple has optimized the living daylights out of this process, and it's less code that you have to write.

Update Links

If crazy, out-of-the-box animations are your thing, then **`UIUpdateLink`** may be what you're looking for.

> ```
> let concatenatedThoughts = """
>
> But wait! How is this any different than `CADisplayLink`? I
> wondered the same. Basically, according to the docs,
> #Features - update link has more of them (like view
> tracking), better performance and battery efficiency and it
> also puts the system in low-latency mode for drawing
> applications.
>
> """
> ```

I imagine this is for applicable for things like custom drawing implementations, complex animations and more. I'm not going to pretend I have any novel examples here, but after a bit of tinkering — I was able to get the examples from the docs working:

```
class TestingViewController: UIViewController {
    let imgView = UIImageView(image: .init(systemName: "rays"))
    var updateLink: UIUpdateLink! = nil

    override func viewDidLoad() {
        super.viewDidLoad()

        imgView.contentMode = .scaleAspectFit
        imgView.frame.size = .init(width: 64, height: 64)
        imgView.frame.origin = .init(x: 100, y: 100)

        updateLink = UIUpdateLink(
            view: imgView,
            actionTarget: self,
            selector: #selector(update)
        )
        updateLink.requiresContinuousUpdates = true
        updateLink.isEnabled = true

        view.addSubview(imgView)
    }

    @objc func update(updateLink: UIUpdateLink,
                      updateInfo: UIUpdateInfo) {
        imgView.center.y = sin(updateInfo.modelTime)
            * 100 + view.bounds.midY
    }
}
```

Which yields:

More Symbol Animations

As always, Cupertino & Friends™ are constantly breathing fresh life into SF Symbols. This year is no different:

- Three new animation styles: Wiggle, breath and rotate. Here's my favorite, **`.breathe`**:

  ```
  override func viewDidLoad() {
    super.viewDidLoad()
    let symbolView = UIImageView(image: .init(systemName: "arrowshape.up
    symbolView.frame = view.bounds.insetBy(dx: 2, dy: 2)
    symbolView.contentMode = .scaleAspectFit
    symbolView.addSymbolEffect(.breathe, options: .repeating, animated:
    view.addSubview(symbolView)
  }
  ```

The result:

- A new behavior, .periodic, which supports a timed delay or a number of times to repeat the animation. Or, you can use **`.repeat(.continuous)`** to keep the party going.
- "Magic replace", which looks so good, smoothly changes badges during replace animations. As far as I can tell, it only works with slashes and badges (going to and from, or vice-versa). But, you can provide a fallback replace behavior if it's not supported. Here's an example from Apple's documentation:

Custom text formatting

Now, we all get to riff on Notes excellent implementation of text formatting:

Even better, it just takes one line of code to opt-in:

```
override func viewDidLoad() {
    super.viewDidLoad()
    let tv = UITextView(frame: view.bounds)
    tv.text = "Format me!"
    tv.allowsEditingTextAttributes = true // This lil' guy is `false` by
    view.addSubview(tv)
}
```

While that is great to offer to developers, what's even better is that we can apparently customize the tools which show here, too. However, these symbols don't appear to be present in the beta one, or they have since-been renamed and I can't seem to track them down. Regardless, it looks like this:

```
tv.textFormattingConfiguration = .init(groups: [
    .group([
        .component(.textColor, .mini)
    ]),
    .group([
        .component(.fontPointSize, .mini)
    ])
])
```

Bonus Points

- You can select dates week by week now using **`UICalendarSelectionWeekOfYear`**:
- SwiftUI and UIKit have unified their gestures, and each one can know about and react to one another.
- **`UICanvasFeedbackGenerator`** can match up haptic events to drawing events. The example Apple gave was a grid-like board, wherein a shape is "snapped" into place on the grid. You could marry haptics along with that experience.
- Now **`UICommand`**, **`UIKeyCommand`** and **`UIAction`** can be invoked by the system on *iPhone*, and that's due to the Mac Mirroring capabilities.
- Lots of sidebar changes along with that spiffy new tab bar, where it floats and morphs into one or the other.

What more can you say? SwiftUI, like the last few years I've written this, is the future. But hey, UIKit, no doubt, is better than it ever has been.

Until next time ✌️

...

Spot an issue, anything to add?
Reach Out.

← @DefaultIfMissing - My Codable Failsafe

W.W.D.C. 2024: The Pregame Quiz →