Sample Code

Implementing modern collection views

Bring compositional layouts to your app and simplify updating your user interface with diffable data sources.

Download

iOS 18.0+ | iPadOS 18.0+ | macOS 11.0+ | Xcode 16.0+

Overview

Collection views let you present a set of data in flexible visual arrangements. This sample app shows how to create various types of layouts and manage data in collection views. The sample focuses on two key technologies:

- Compositional layout, a type of collection view layout that's composable, flexible, and fast, letting you build any kind of visual arrangement for your content.
- Diffable data source, a specialized type of data source that provides the behavior you need to simply and efficiently manage updates to your collection view's data and user interface.

Configure the sample code project

To run the sample code project in Xcode, first choose whether to view the examples in iOS or macOS.

To view examples in iOS:

1. Choose the Modern Collection Views target.
2. In the Scheme menu, choose an iOS simulator to run the app.

To view examples in macOS:

1. Choose the Modern Collection Views Mac target.
2. In the Scheme menu, choose My Mac.
3. In Build Settings for the target, under Signing & Capabilities > Signing Certificate, choose Sign to Run Locally.
4. Run the app, and navigate to the examples from the Example menu.

The code examples shown here are from the iOS target, but you can find macOS-equivalent examples in the `.swift` files for the macOS target.

Create a grid layout

The Grid example shows how to create a grid layout by using fractional sizing to make a row of five equally sized items. It creates a horizontal group with items that are each 20% of the width of the group by using `.fractionalWidth(0.2)`. Each row of five items is repeated multiple times in a single section to create a grid.

```
let itemSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(0.2),
                                      heightDimension: .fractionalHeight(1.0))
let item = NSCollectionLayoutItem(layoutSize: itemSize)

let groupSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                       heightDimension: .fractionalWidth(0.2))
let group = NSCollectionLayoutGroup.horizontal(layoutSize: groupSize,
                                               subitems: [item])

let section = NSCollectionLayoutSection(group: group)

let layout = UICollectionViewCompositionalLayout(section: section)
return layout
```

Add spacing around items

The Inset Items Grid example builds on the layout from the Grid example, showing how to add spacing around items by using the `contentInsets` property. Here, this property applies even spacing around the edges of each item.

```
let itemSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(0.2),
                                      heightDimension: .fractionalHeight(1.0))
let item = NSCollectionLayoutItem(layoutSize: itemSize)
```

```
item.contentInsets = NSDirectionalEdgeInsets(top: 5, leading: 5, bottom: 5, trailir
```

Create a column layout

The Two-Column Grid example shows how to create a two-column layout by making a group with the exact number of items specified in the `count` parameter of `horizontal(layoutSize:subitem:count:)`. This approach simplifies specifying exactly how many items a group contains. In this case, the `count` parameter takes precedence over `itemSize`, and item size is computed automatically to fit the specified number of items.

```
let itemSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                      heightDimension: .fractionalHeight(1.0))
let item = NSCollectionLayoutItem(layoutSize: itemSize)

let groupSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                       heightDimension: .absolute(44))
let group = NSCollectionLayoutGroup.horizontal(layoutSize: groupSize, repeatingSubit
let spacing = CGFloat(10)
group.interItemSpacing = .fixed(spacing)
```

Display distinct layouts per section

The Distinct Sections example shows how to display different layout arrangements in different sections of the same collection view layout. Creating a layout with different sections requires a compositional layout with a section provider. The code in the section provider accesses the section's index (`sectionIndex`) to determine which section it's configuring, and displays a different layout for each section.

```
let layout = UICollectionViewCompositionalLayout { (sectionIndex: Int,
    layoutEnvironment: NSCollectionLayoutEnvironment) -> NSCollectionLayoutSection?

    guard let sectionLayoutKind = SectionLayoutKind(rawValue: sectionIndex) else { r
    let columns = sectionLayoutKind.columnCount

    // The group auto-calculates the actual item width to make
    // the requested number of columns fit, so this widthDimension is ignored.
    let itemSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                          heightDimension: .fractionalHeight(1.0))
    let item = NSCollectionLayoutItem(layoutSize: itemSize)
    item.contentInsets = NSDirectionalEdgeInsets(top: 2, leading: 2, bottom: 2, trai
```

```
    let groupHeight = columns == 1 ?
        NSCollectionLayoutDimension.absolute(44) :
        NSCollectionLayoutDimension.fractionalWidth(0.2)
    let groupSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                           heightDimension: groupHeight)
    let group = NSCollectionLayoutGroup.horizontal(layoutSize: groupSize, repeatingS

    let section = NSCollectionLayoutSection(group: group)
    section.contentInsets = NSDirectionalEdgeInsets(top: 20, leading: 20, bottom: 20
    return section
}
return layout
```

Display distinct layouts in different environments

The Adaptive Sections example shows how to create a layout that adapts to the environment it's displayed in. In this example, the number of columns shown changes based on the available screen size. Creating a layout that adapts to a new environment requires a compositional layout with a section provider. The code in the section provider accesses the amount of available space in the current layout environment (`layoutEnvironment.container.effectiveContentSize`), and displays a different number of columns based on the available width.

```
let layout = UICollectionViewCompositionalLayout {
    (sectionIndex: Int, layoutEnvironment: NSCollectionLayoutEnvironment) -> NSColle
    guard let layoutKind = SectionLayoutKind(rawValue: sectionIndex) else { return n

    let columns = layoutKind.columnCount(for: layoutEnvironment.container.effectiveC

    let itemSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(0.2),
                                          heightDimension: .fractionalHeight(1.0))
    let item = NSCollectionLayoutItem(layoutSize: itemSize)
    item.contentInsets = NSDirectionalEdgeInsets(top: 2, leading: 2, bottom: 2, trai

    let groupHeight = layoutKind == .list ?
        NSCollectionLayoutDimension.absolute(44) : NSCollectionLayoutDimension.fract
    let groupSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                           heightDimension: groupHeight)
    let group = NSCollectionLayoutGroup.horizontal(layoutSize: groupSize, repeatingS
    let section = NSCollectionLayoutSection(group: group)
    section.contentInsets = NSDirectionalEdgeInsets(top: 20, leading: 20, bottom: 20
    return section
}
```

```
return layout
```

Add badges to items

The Item Badges example shows how to add supplementary views like badges to the items in a collection view. It creates and adds a badge to the top trailing corner of each item by creating a supplementary item for the badge, and passing in that supplementary item when creating the item itself. The data source configures each badge in its `supplementaryViewProvider`.

```
let badgeAnchor = NSCollectionLayoutAnchor(edges: [.top, .trailing], fractionalOffse
let badgeSize = NSCollectionLayoutSize(widthDimension: .absolute(20),
                                       heightDimension: .absolute(20))
let badge = NSCollectionLayoutSupplementaryItem(
    layoutSize: badgeSize,
    elementKind: ItemBadgeSupplementaryViewController.badgeElementKind,
    containerAnchor: badgeAnchor)

let itemSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(0.25),
                                      heightDimension: .fractionalHeight(1.0))
let item = NSCollectionLayoutItem(layoutSize: itemSize, supplementaryItems: [badge])
item.contentInsets = NSDirectionalEdgeInsets(top: 5, leading: 5, bottom: 5, trailing
```

Add headers and footers to sections

The Section Headers/Footers example shows how to add headers and footers to each section of the collection view. It creates boundary supplementary items to represent the header and the footer, and sets them as the section's `boundarySupplementaryItems`.

```
let headerFooterSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                              heightDimension: .estimated(44))
let sectionHeader = NSCollectionLayoutBoundarySupplementaryItem(
    layoutSize: headerFooterSize,
    elementKind: SectionHeadersFootersViewController.sectionHeaderElementKind, alignr
let sectionFooter = NSCollectionLayoutBoundarySupplementaryItem(
    layoutSize: headerFooterSize,
    elementKind: SectionHeadersFootersViewController.sectionFooterElementKind, alignr
section.boundarySupplementaryItems = [sectionHeader, sectionFooter]
```

The example uses supplementary registrations for the header and footer to configure their content and appearance.

```
let headerRegistration = UICollectionView.SupplementaryRegistration
<TitleSupplementaryView>(elementKind: SectionHeadersFootersViewController.sectionHea
    (supplementaryView, string, indexPath) in
    supplementaryView.label.text = "\(string) for section \(indexPath.section)"
    supplementaryView.backgroundColor = .lightGray
    supplementaryView.layer.borderColor = UIColor.black.cgColor
    supplementaryView.layer.borderWidth = 1.0
}
```

The collection view uses these supplementary registrations to dequeue the configured headers and footers in the diffable data source's `supplementaryViewProvider`.

```
dataSource.supplementaryViewProvider = { (view, kind, index) in
    return self.collectionView.dequeueConfiguredReusableSupplementary(
        using: kind == SectionHeadersFootersViewController.sectionHeaderElementKind
}
```

Pin section headers to sections

The Pinned Section Headers example shows how to pin a section header to its section. This way, the header shows while any portion of the section it's attached to is visible during scrolling, and the footer stays in place. This example sets the header's `pinToVisibleBounds` property to `true` and increases its `zIndex` to a value greater than 1 so the header appears on top of the section during scrolling.

```
let sectionHeader = NSCollectionLayoutBoundarySupplementaryItem(
    layoutSize: NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                       heightDimension: .estimated(44)),
    elementKind: PinnedSectionHeaderFooterViewController.sectionHeaderElementKind,
    alignment: .top)
let sectionFooter = NSCollectionLayoutBoundarySupplementaryItem(
    layoutSize: NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                       heightDimension: .estimated(44)),
    elementKind: PinnedSectionHeaderFooterViewController.sectionFooterElementKind,
    alignment: .bottom)
sectionHeader.pinToVisibleBounds = true
sectionHeader.zIndex = 2
section.boundarySupplementaryItems = [sectionHeader, sectionFooter]
```

The example uses supplementary registrations for the header and footer to configure their content and appearance. The collection view uses these supplementary registrations to dequeue the configured headers and footers in the diffable data source's `supplementaryViewProvider`.

```
dataSource.supplementaryViewProvider = { (view, kind, index) in
    return self.collectionView.dequeueConfiguredReusableSupplementary(
        using: kind == PinnedSectionHeaderFooterViewController.sectionHeaderElementK
}
```

Decorate sections with backgrounds

The Section Background Decoration example shows how to distinguish sections by adding section backgrounds. It creates a section background by making a decoration item using `background(elementKind:)`. It then sets that decoration item as the section's `decorationItems` property.

```
let sectionBackgroundDecoration = NSCollectionLayoutDecorationItem.background(
    elementKind: SectionDecorationViewController.sectionBackgroundDecorationElementK
sectionBackgroundDecoration.contentInsets = NSDirectionalEdgeInsets(top: 5, leading:
section.decorationItems = [sectionBackgroundDecoration]
```

The following code then registers the background view with the layout by using `register(_:forDecorationViewOfKind:)`.

```
let layout = UICollectionViewCompositionalLayout(section: section)
layout.register(
    SectionBackgroundDecorationView.self,
    forDecorationViewOfKind: SectionDecorationViewController.sectionBackgroundDecora
return layout
```

Create custom layouts by nesting groups

The Nested Groups example shows how to create flexible layout arrangements by nesting groups inside of other groups. It creates a vertical group with two items, and combines the vertical group with an item in a horizontal parent group.

```
let leadingItem = NSCollectionLayoutItem(
    layoutSize: NSCollectionLayoutSize(widthDimension: .fractionalWidth(0.7),
                                       heightDimension: .fractionalHeight(1.0)))
```

```
leadingItem.contentInsets = NSDirectionalEdgeInsets(top: 10, leading: 10, bottom: 10

let trailingItem = NSCollectionLayoutItem(
    layoutSize: NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                       heightDimension: .fractionalHeight(0.3)))
trailingItem.contentInsets = NSDirectionalEdgeInsets(top: 10, leading: 10, bottom: 1
let trailingGroup = NSCollectionLayoutGroup.vertical(
    layoutSize: NSCollectionLayoutSize(widthDimension: .fractionalWidth(0.3),
                                        heightDimension: .fractionalHeight(1.0)),
    repeatingSubitem: trailingItem,
    count: 2)
let nestedGroup = NSCollectionLayoutGroup.horizontal(
    layoutSize: NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                       heightDimension: .fractionalHeight(0.4)),
    subitems: [leadingItem, trailingGroup])
```

Scroll sections horizontally

The Orthogonal Sections example shows how to create a section that scrolls horizontally in an otherwise vertically scrolling layout. Setting a section's `orthogonalScrollingBehavior` property to a value other than `UICollectionLayoutSectionOrthogonalScrollingBehavior.none` causes the section to lay out its contents perpendicular to the main layout axis. In this case, because the layout scrolls vertically by default, the section scrolls horizontally.

```
section.orthogonalScrollingBehavior = .continuous
```

Choose horizontal scrolling and paging behavior

The Orthogonal Section Behaviors example shows each of the options for `UICollectionLayoutSectionOrthogonalScrollingBehavior`. Each section of the layout demonstrates a different orthogonal scrolling behavior, showcasing the differences between the scrolling and paging options. In this case, because the layout scrolls vertically by default, the sections themselves scroll horizontally.

```
case continuous, continuousGroupLeadingBoundary, paging, groupPaging, groupPagingCer
func orthogonalScrollingBehavior() -> UICollectionLayoutSectionOrthogonalScrollingBe
    switch self {
    case .none:
        return UICollectionLayoutSectionOrthogonalScrollingBehavior.none
    case .continuous:
        return UICollectionLayoutSectionOrthogonalScrollingBehavior.continuous
```

```
    case .continuousGroupLeadingBoundary:
        return UICollectionLayoutSectionOrthogonalScrollingBehavior.continuousGroupL
    case .paging:
        return UICollectionLayoutSectionOrthogonalScrollingBehavior.paging
    case .groupPaging:
        return UICollectionLayoutSectionOrthogonalScrollingBehavior.groupPaging
    case .groupPagingCentered:
        return UICollectionLayoutSectionOrthogonalScrollingBehavior.groupPagingCente
    }
}
```

Update data in a collection view

The Mountains Search example shows how to update the data and the user interface in a collection view when a user filters the data. It shows a list of mountain names, and allows the user to type text into a search bar to filter based on the mountain names.

This collection view, `mountainsCollectionView`, contains a single section with items created from a list of raw data about each mountain. This example encapsulates each piece of that data in a `Mountain` structure, defined in `MountainsController`. To manage the data, this example uses a diffable data source that contains a single section and items of type `Mountain`. When that diffable data source is created, it connects to `mountainsCollectionView` and registers a cell type of `LabelCell` to display the name of the mountain in the collection view. Then, it configures that cell with the name of the mountain.

The `performQuery(with:)` method updates the data and the user interface. The method takes the currently typed filter text and generates a list of mountains that contain that text in their names. Then, it constructs a representation of the newly filtered data using a snapshot. The snapshot contains the same single section as before, but now, instead of containing items representing every mountain, it only contains the filtered mountains.

The method then calls `apply(_:animatingDifferences:completion:)`, applying the data from the snapshot to the diffable data source. The diffable data source stores the data from the snapshot as the new state of data, calculates the difference between the previous state and the new state, and triggers the user interface to display the new state.

```
func performQuery(with filter: String?) {
    let mountains = mountainsController.filteredMountains(with: filter).sorted { $0.

    var snapshot = NSDiffableDataSourceSnapshot<Section, MountainsController.Mountai
    snapshot.appendSections([.main])
    snapshot.appendItems(mountains)
    dataSource.apply(snapshot, animatingDifferences: true)
```

```
}
```

Update data in multiple sections

The Settings: Wi-Fi example shows how to update the data and the user interface in a table view that uses multiple kinds of sections and items. It recreates the Wi-Fi page in iOS Settings, letting the user toggle the Wi-Fi switch on and off to view the available and current Wi-Fi networks.

The `updateUI(animated:)` method determines which sections and items to display based on whether Wi-Fi is enabled. If Wi-Fi is disabled, the method adds only the `.config` section and its items to the snapshot. If Wi-Fi is enabled, the method adds the `.networks` section and its items as well.

```
let configItems = configurationItems.filter { !($0.type == .currentNetwork && !contr

currentSnapshot = NSDiffableDataSourceSnapshot<Section, Item>()

currentSnapshot.appendSections([.config])
currentSnapshot.appendItems(configItems, toSection: .config)

if controller.wifiEnabled {
    let sortedNetworks = controller.availableNetworks.sorted { $0.name < $1.name }
    let networkItems = sortedNetworks.map { Item(network: $0) }
    currentSnapshot.appendSections([.networks])
    currentSnapshot.appendItems(networkItems, toSection: .networks)
}

self.dataSource.apply(currentSnapshot, animatingDifferences: animated)
```

Update data incrementally

The Insertion Sort Visualization example shows how to update data incrementally, displaying the visible progress as the data changes from an initial state to a final state. It shows rows of color swatches, originally in a random order, that it then iteratively sorts into spectral order.

The key difference between this example and the other diffable data source examples is that this example doesn't create an empty snapshot to update the state of the data. Instead, the `performSortStep()` method retrieves the current state of the collection view's data by using `dataSource.snapshot()`. Then, it modifies only one part of that snapshot to perform the visual sorting step by step.

```
// Get the current state of the UI from the data source.
var updatedSnapshot = dataSource.snapshot()

// For each section, if needed, step through and perform the next sorting step.
updatedSnapshot.sectionIdentifiers.forEach {
    let section = $0
    if !section.isSorted {

        // Step the sort algorithm.
        section.sortNext()
        let items = section.values

        // Replace the items for this section with the newly sorted items.
        updatedSnapshot.deleteItems(items)
        updatedSnapshot.appendItems(items, toSection: section)

        sectionCountNeedingSort += 1
    }
}
```

Create a simple list layout

The Simple List example shows how to create a basic list layout that adapts to any screen size. It creates a configuration with one of the system-defined list appearances. Then, it creates a list layout by passing the configuration to `list(using:)`. This approach generates a compositional layout with a single section styled as a list.

```
let config = UICollectionLayoutListConfiguration(appearance: .insetGrouped)
return UICollectionViewCompositionalLayout.list(using: config)
```

Choose a list appearance

The List Appearances example shows each of the options for `UICollectionLayoutListConfiguration.Appearance`. Tapping the name of the appearance in the navigation bar changes the list to another appearance, showcasing each of the list styles. Each list uses the `UICollectionLayoutListConfiguration.HeaderMode.firstItemInSection` header mode, which makes each section of the list expandable and collapsible.

```
var config = UICollectionLayoutListConfiguration(appearance: self.appearance)
config.headerMode = .firstItemInSection
```

Customize list cells

The List with Custom Cells example shows how to configure a custom list cell subclass. The example focuses on `CustomListCell`, a custom subclass of `UICollectionViewListCell` that combines several kinds of subviews into one cell. It sets the appearance and content of these views using content configurations.

The `updateConfiguration(using:)` method sets up the cell's initial appearance and content. The system calls this method every time the cell's configuration state changes to update the cell's appearance for that new state. To configure the list content view, it fetches the default list content configuration for the current state.

```
var content = defaultListContentConfiguration().updated(for: state)
```

Then, it customizes the configuration's values and assigns that configuration to the `configuration` property of `listContentView`.

For the image view and label, the `updateConfiguration(using:)` method fetches a default value cell configuration for the current state and stores it in `valueConfiguration`. It copies the preconfigured default styling and metrics from this configuration into the custom views to ensure consistency with the system styles.

```
categoryIconView.tintColor = valueConfiguration.imageProperties.resolvedTintColor(fo
categoryIconView.preferredSymbolConfiguration = .init(font: valueConfiguration.secor
```

To register the custom cell subclass with the collection view, this example uses a cell registration. The cell registration configures each cell with the data from its corresponding item. It also adds a disclosure indicator cell accessory to the cell.

```
let cellRegistration = UICollectionView.CellRegistration<CustomListCell, Item> { (ce
    cell.updateWithItem(item)
    cell.accessories = [.disclosureIndicator()]
}
```

The diffable data source uses that cell registration when it dequeues the cell.

```
return collectionView.dequeueConfiguredReusableCell(using: cellRegistration, for: ir
```

Build a layout with multiple section types

The Emoji Explorer example shows how to create a compositional layout with multiple types of sections. The example contains an orthogonally scrolling section, an outline section, and a list section in one compositional layout. The `createLayout()` method defines a section provider to supply each section.

For the top section, it enables horizontal scrolling by setting the `orthogonalScrollingBehavior` property.

```
section.orthogonalScrollingBehavior = .continuousGroupLeadingBoundary
```

The outline section uses the `UICollectionLayoutListConfiguration.Appearance.sidebar` list appearance. It's populated using a section snapshot to create a hierarchical data structure.

```
let rootItem = Item(title: String(describing: category), hasChildren: true)
outlineSnapshot.append([rootItem])
let outlineItems = category.emojis.map { Item(emoji: $0) }
outlineSnapshot.append(outlineItems, to: rootItem)
```

The list section uses the `UICollectionLayoutListConfiguration.Appearance.insetGrouped` list appearance. The configuration for this section adds a swipe action to each cell that lets the cell be marked as Favorite.

```
configuration.leadingSwipeActionsConfigurationProvider = { [weak self] (indexPath) i
    guard let self = self else { return nil }
    guard let item = self.dataSource.itemIdentifier(for: indexPath) else { return ni
    return self.leadingSwipeActionConfigurationForListCellItem(item)
}
```

Each section has a corresponding cell registration to configure its own type of cell. The collection view uses those registrations to dequeue the configured cells to display in each section.

```
switch section {
case .recents:
    return collectionView.dequeueConfiguredReusableCell(using: gridCellRegistration,
case .list:
    return collectionView.dequeueConfiguredReusableCell(using: listCellRegistration,
case .outline:
    if item.hasChildren {
        return collectionView.dequeueConfiguredReusableCell(using: outlineHeaderCell
```

```
    } else {
        return collectionView.dequeueConfiguredReusableCell(using: outlineCellRegist
    }
```

Create a value cell list

The Emoji Explorer - List example shows how to create a list with cells that use the value cell style. It applies a content configuration with the default value cell style to each of the cells in the list.

```
var contentConfiguration = UIListContentConfiguration.valueCell()
contentConfiguration.text = emoji.text
contentConfiguration.secondaryText = String(describing: emoji.category)
cell.contentConfiguration = contentConfiguration
```

See Also

Data

{} Updating collection views using diffable data sources

Streamline the display and update of data in a collection view using a diffable data source that contains identifiers.

{} Building high-performance lists and collection views

Improve the performance of lists and collections in your app with prefetching and image preparation.

`class` `UICollectionViewDiffableDataSource`

The object you use to manage data and provide cells for a collection view.

`protocol` `UICollectionViewDataSource`

The methods adopted by the object you use to manage data and provide cells for a collection view.

`protocol` `UICollectionViewDataSourcePrefetching`

A protocol that provides advance warning of the data requirements for a collection view, allowing the triggering of asynchronous data load operations.

`struct` `NSDiffableDataSourceSnapshot`

A representation of the state of the data in a view at a specific point in time.

`struct` `NSDiffableDataSourceSectionSnapshot`

A representation of the state of the data in a layout section at a specific point in time.

`class` `UIRefreshControl`

A standard control that can initiate the refreshing of a scroll view's contents.