

[UIKit](#) / [...](#) / [Collection views](#) / Updating collection views using diffable data sources

Sample Code

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.

[Download](#)

iOS 15.0+ | iPadOS 15.0+ | Mac Catalyst 15.0+ | Xcode 13.2+

Overview

A *collection view* presents data in the form of sections and items, and an app that displays data in a collection view inserts those sections and items into the view. The app may also need to handle deleting or moving sections and items. For instance, the sample app in this project displays recipes in a collection view, and people using the app can add and delete recipes, and mark recipes as favorites. To support these actions, the sample app handles inserting, deleting, moving, and updating data within a collection view.

When populating a collection view in an app, you can create a custom data source that adopts the [UICollectionViewDataSource](#) protocol. To keep the information in the collection view current, you determine what data changed and perform a batch update based on those changes, a process that requires careful coordination of inserts, deletes, and moves.

To avoid the complexity of that process, the sample app uses a [UICollectionViewDiffableDataSource](#) object. A *diffable data source* stores a list of section and item *identifiers*, which represents the identity of each section and item contained in a collection view. These identifiers are stable, meaning they don't change. In contrast, a custom data source that conforms to [UICollectionViewDataSource](#) uses *indices* and *index paths*, which aren't stable. They represent the location of sections and items, which can change as the data source adds, removes, and rearranges the contents of a collection view. However, with identifiers a diffable data source can refer to a section or item without knowledge of its location within a collection view.

Note

This sample uses collection views to display data, but the concepts covered in this sample apply to table views as well. For more information about using a diffable data source with a table view, see [UITableViewDiffableDataSource](#).

To use a value as an identifier, its data type must conform to the [Hashable](#) protocol. Hashing allows data collections such as [Set](#), [Dictionary](#), and snapshots — instances of [NSDiffableDataSourceSnapshot](#) and [NSDiffableDataSourceSectionSnapshot](#) — to use values as keys, providing quick and efficient lookups. Hashable types also conform to the [Equatable](#) protocol, so your identifiers must properly implement equality. For more information, see [Equatable](#).

Because identifiers are hashable and equatable, a diffable data source can determine the differences between its current snapshot and another snapshot. Then it can insert, delete, and move sections and items within a collection view for you based on those differences, eliminating the need for custom code that performs batch updates.

Important

Two identifiers that are equal must always have the same hash value. However, the converse isn't true; two values with the same hash value aren't required to be equal. This situation is called a *hash collision*. To increase efficiency, try to ensure that unequal identifiers have different hash values. The occasional hash collision is okay when it's unavoidable, but keep the number of collisions to a minimum. Otherwise, the performance of lookups in the data collection may suffer.

Define the diffable data source

In this sample project, `RecipeListViewController` displays a list of recipes in a collection view. Before the controller can display the recipes, it defines an instance variable to store a diffable data source.

```
private var recipeListDataSource: UICollectionViewDiffableDataSource<RecipeListSection, Recipe.ID>
```

`RecipeListViewController` declares `recipeListDataSource` with `RecipeListSection` as the section identifier type, and `Recipe.ID` as the item identifier type. These identifier types tell the data source the type of values it contains.

For the section identifier type, `recipeListDataSource` uses `RecipeListSection`, an enumeration with a raw value of type [Int](#) (in Swift, `Int` is hashable). Each enumeration case

identifies a section of the collection view. In the sample, there's only one section, `main`, which displays a list of recipes.

```
private enum RecipeListSection: Int {  
    case main  
}
```

For the item identifier type, `recipeListDataSource` uses `Recipe.ID`. This type comes from the `Recipe` structure, defined as:

```
struct Recipe: Identifiable, Codable {  
    var id: Int  
    var title: String  
    var prepTime: Int    // In seconds.  
    var cookTime: Int    // In seconds.  
    var servings: String  
    var ingredients: String  
    var directions: String  
    var isFavorite: Bool  
    var collections: [String]  
    fileprivate var addedOn: Date? = Date()  
    fileprivate var imageNames: [String]  
}
```

This structure conforms to the `Identifiable` protocol, which requires the structure to include an `id` property. By conforming to `Identifiable`, the `Recipe` structure automatically exposes the associated type `ID`, which is a type determined based on the declaration of the `id` property in the structure. And because this type must be hashable, the sample app can use `Recipe.ID` as the item identifier type.

Note

The `Recipe` structure doesn't conform to the `Hashable` protocol. The structure doesn't have to be hashable because the items stored in the diffable data source and the snapshots are recipe *identifiers* (`Recipe.ID` values the backing data store provides for each recipe), not complete recipe structures.

Using the `Recipe.ID` as the item identifier type for the `recipeListDataSource` means that the data source, and any snapshots applied to it, contains only `Recipe.ID` values and not the

complete recipe data. This approach optimizes the diffable data source for peak performance when displaying recipes in a collection view because the identifier type is a simple, hashable type.

Configure the diffable data source

Before populating a collection view with data from a diffable data source, the sample app configures the data source. The app creates an instance of `UICollectionViewDiffableDataSource` and sets its *cell provider*, a closure that configures and returns a cell for the collection view.

`RecipeListViewController` configures `recipeListDataSource` in a helper method named `configureDataSource()`. The view controller calls this method in its `viewDidLoad()` method.

The `configureDataSource()` method creates a cell registration and provides a handler closure that configures each cell with data from a recipe. The closure receives an instance of `Recipe`, which it uses to configure the cell.

Note

The item type for a cell registration doesn't have to match the item identifier type that the diffable data source uses.

Next, `configureDataSource()` creates an instance of `UICollectionViewDiffableDataSource` and defines the cell provider closure. The closure receives the identifier of a recipe. It then retrieves the recipe from the backing data store (using the identifier) and passes the recipe structure to the cell registration's handler closure.

```
private func configureDataSource() {
    // Create a cell registration that the diffable data source will use.
    let recipeCellRegistration = UICollectionView.CellRegistration<UICollectionView
    var contentConfiguration = UIListContentConfiguration.subtitleCell()
    contentConfiguration.text = recipe.title
    contentConfiguration.secondaryText = recipe.subtitle
    contentConfiguration.image = recipe.smallImage
    contentConfiguration.imageProperties.cornerRadius = 4
    contentConfiguration.imageProperties.maximumSize = CGSize(width: 60, height:
    cell.contentConfiguration = contentConfiguration

    if recipe.isFavorite {
        let image = UIImage(systemName: "heart.fill")
    }
}
```

```

        let accessoryConfiguration = UITableViewCellAccessory.CustomViewConfiguration(customView: tableView.placementImage)

        cell.accessories = [.customView(configuration: accessoryConfiguration)]
    } else {
        cell.accessories = []
    }
}

// Create the diffable data source and its cell provider.
recipeListDataSource = UICollectionViewDiffableDataSource(collectionView: collectionView, indexPath, identifier -> UICollectionViewCell in {
    // `identifier` is an instance of `Recipe.ID`. Use it to
    // retrieve the recipe from the backing data store.
    let recipe = datastore.recipe(with: identifier)!
    return collectionView.dequeueConfiguredReusableCell(using: recipeCellRegistrar)
})
}

```

Load the diffable data source with identifiers

With the diffable data source configured, `RecipeListViewController` calls its helper method `loadRecipeData()` to perform an initial load of data into the data source, which in turn populates a collection view with recipes. This method retrieves a list of recipe identifiers and creates an instance of `NSDiffableDataSourceSnapshot`. Then it adds the main section and recipe identifiers to the snapshot. Lastly, the method calls `applySnapshotUsingReloadData(_:)` to apply the snapshot to the data source, resetting the collection view to reflect the state of the data in the snapshot without computing a diff or animating the changes.

```

private func loadRecipeData() {
    // Retrieve the list of recipe identifiers determined based on a
    // selected sidebar item such as All Recipes or Favorites.
    guard let recipeIds = recipeSplitViewController.selectedRecipes?.recipeIds()
    else { return }

    // Update the collection view by adding the recipe identifiers to
    // a new snapshot, and apply the snapshot to the diffable data source.
    var snapshot = NSDiffableDataSourceSnapshot<RecipeListSection, Recipe.ID>()
    snapshot.appendSections([.main])
    snapshot.appendItems(recipeIds, toSection: .main)
    recipeListDataSource.applySnapshotUsingReloadData(snapshot)
}

```

Important

Each item identifier must be unique within a snapshot. As a result, an item identifier can't appear in multiple locations within a snapshot. The same is true of section identifiers; they must be unique and can't exist in multiple places within a snapshot.

Insert, delete, and move items

People using the sample app can make two types of changes to the recipe data:

- Changes to the collection of data itself, like adding or removing recipes, or reordering them.
- Changes to the properties of existing items, like changing the name of a recipe or marking one as a favorite.

To handle changes to a data collection, the app creates a new snapshot that represents the current state of the data collection and applies it to the diffable data source. The data source compares its current snapshot with the new snapshot to determine the changes. Then it performs the necessary inserts, deletes, and moves into the collection view based on those changes.

While a diffable data source can determine the changes between its current snapshot and a new one, it doesn't monitor the data collection for changes. Instead, it's the responsibility of the app to detect data changes and tell the diffable data source about those changes, by applying a new snapshot.

Note

An app can use different mechanisms, such as [NotificationCenter](#) and [Combine](#), to report data changes to other parts of the app. This sample uses [NotificationCenter](#).

To inform other parts of the app that the list of recipes changed — for instance, after someone adds or removes a recipe — the sample uses a notification center to send a `selectedRecipesDidChange` notification. To receive the notification, `RecipeListViewController` adds a notification observer with `selectedRecipesDidChange(_:)` as its selector.

```
NotificationCenter.default.addObserver(  
    self,  
    selector: #selector(selectedRecipesDidChange(_:)),  
    name: .selectedRecipesDidChange,  
    object: nil  
)
```

`selectedRecipesDidChange(_:)` is similar to `loadRecipeData()` but it uses `apply(_:animatingDifferences:)` to apply the list of selected recipe identifiers that the notification provides instead of using `applySnapshotUsingReloadData(_:)`. The `apply(_:animatingDifferences:)` method performs incremental updates to the collection view instead of entirely resetting the data displayed. And because `animatingDifferences` is `true`, the collection view animates the changes as they appear.

@objc

```
private func selectedRecipesDidChange(_ notification: Notification) {
    // Create a snapshot of the selected recipe identifiers from the notification's
    // `userInfo` dictionary, and apply it to the diffable data source.
    guard
        let userInfo = notification.userInfo,
        let selectedRecipeIds = userInfo[NotificationKeys.selectedRecipeIds] as? [Recipe.ID]
    else { return }

    var snapshot = NSDiffableDataSourceSnapshot<RecipeListSection, Recipe.ID>()
    snapshot.appendSections([.main])
    snapshot.appendItems(selectedRecipeIds, toSection: .main)
    recipeListDataSource.apply(snapshot, animatingDifferences: true)

    // The design of this sample app makes it possible for the selected
    // recipe displayed in the secondary (detail) view controller to exist
    // in the new snapshot but not exist in the collection view prior to
    // applying the snapshot. For instance, while displaying the list of
    // favorite recipes, a person can unfavorize the selected recipe by tapping
    // the `isFavorite` button. This removes the selected recipe from the
    // favorites list. Tap the button again and the recipe reappears in the
    // list. In this scenario, the app needs to re-select the recipe so it
    // appears as selected in the collection view.
    selectRecipeIfNeeded()
}
```

Update existing items

To handle changes to the properties of an existing item, an app retrieves the current snapshot from the diffable data source and calls either `reconfigureItems(_:)` or `reloadItems(_:)` on the snapshot. Then it applies the snapshot to the diffable data source, which updates the display of the specified items.

Again, the app, not the diffable data source, detects the data changes.

To tell others parts of the app about a change to a recipe — for instance, when a person marks a recipe as a favorite — the sample sends a `recipeDidChange` notification. `RecipeListView` Controller receives the notification using an observer with `recipeDidChange(_:)` as the selector.

```
NotificationCenter.default.addObserver(
    self,
    selector: #selector(recipeDidChange(_:)),
    name: .recipeDidChange,
    object: nil
)
```

The `recipeDidChange` notification indicates that data for a single recipe changed. Because only one recipe changed, there's no need to update the entire list of recipes shown in the collection view. Instead, the sample only updates the cell that displays the recipe that changed. For instance, when a person marks a recipe as a favorite, an icon of a heart appears beside that recipe. And when the person unmarks the recipe as a favorite, the heart disappears.

To update the cell with the latest recipe data, the `recipeDidChange(_:)` method confirms that the diffable data source contains the recipe identifier that the notification provides. Then the method retrieves the current snapshot from the data source and calls `reconfigureItems(_:)`, passing in the recipe identifier. This call tells the data source to update the data displayed in the cell identified by the recipe identifier. Finally, `recipeDidChange(_:)` applies the updated snapshot to the data source.

```
@objc
private func recipeDidChange(_ notification: Notification) {
    guard
        // Get `recipeId` from from the `userInfo` dictionary.
        let userInfo = notification.userInfo,
        let recipeId = userInfo[NotificationKeys.recipeId] as? Recipe.ID,
        // Confirm that the data source contains the recipe.
        recipeListDataSource.indexPath(for: recipeId) != nil
    else { return }

    // Get the diffable data source's current snapshot.
    var snapshot = recipeListDataSource.snapshot()
    // Update the recipe's data displayed in the collection view.
    snapshot.reconfigureItems([recipeId])
    recipeListDataSource.apply(snapshot, animatingDifferences: true)
}
```

The diffable data source compares the updated snapshot to its current snapshot and applies the difference — in this instance, a request to reconfigure the item that displays the recipe that changed. To fulfill the request, the data source invokes its cell provider closure, which retrieves the updated recipe and configures the cell with the latest recipe data. And because `animatingDifferences` is `true` when applying the snapshot, the collection view animates the visual change of the cell by showing or hiding the heart icon.

Populate snapshots with lightweight data structures

An alternative approach to storing identifiers involves populating diffable data sources and snapshots with lightweight data structures. While the data structure approach is convenient and can be a good fit in some circumstances — like for quick prototyping, or displaying a collection of static items with properties that don't change — it carries significant limitations and tradeoffs. For instance, the `Hashable` and `Equatable` implementations must incorporate all properties of the structure that can change. Any changes to the data in the structure cause it to no longer be equal to the previous version, which the diffable data source uses to determine what changed when applying a new snapshot.

The sample uses this approach to show items in a sidebar. In `SidebarViewController`, the custom structure `SidebarItem` defines the properties of a sidebar item, which are `title` and `type`.

```
private struct SidebarItem: Hashable {
    let title: String
    let type: SidebarItemType

    enum SidebarItemType {
        case standard, collection, expandableHeader
    }
}
```

The combination of these properties determine the hashing value for each sidebar item, and because the property values don't change, populating the snapshot with this `SidebarItem` structure instead of identifiers is an acceptable use case.

```
private func createSnapshotOfStandardItems() -> NSDiffableDataSourceSectionSnapshot<SidebarItem> {
    let items = [
        SidebarItem(title: StandardSidebarItem.all.rawValue, type: .standard),
        SidebarItem(title: StandardSidebarItem.favorites.rawValue, type: .standard),
        SidebarItem(title: StandardSidebarItem.recents.rawValue, type: .standard)
    ]
}
```

```
return createSidebarItemSnapshot(.standardItems, items: items)
```

The downside of this approach is that the diffable data source can no longer track identity. Any time an existing item changes, the diffable data source sees the change as a delete of the old item and an insert of a new item. As a result, the collection view loses important state tied to the item. For instance, a selected item becomes unselected when any property of the item changes because, from the diffable data source's perspective, the app deleted the item and added a new one to take its place.

Also, if `animatingDifferences` is `true` when applying the snapshot, every change requires the process of animating out the old cell and animating in a new cell, which can be detrimental to performance and cause loss of UI state, including animations, within the cell.

Additionally, this strategy precludes using the `reconfigureItems(_:)` or `reloadItems(_:)` methods when populating a snapshot with data structures, because those methods require the use of proper identifiers. The only mechanism to update the data for existing items is to apply a new snapshot containing the new data structures, which causes the diffable data source to perform a delete and an insert for each changed item.

Storing data structures directly into diffable data sources and snapshots isn't a robust solution for many real-world use cases because the data source loses the ability to track identity. Only use this approach for simple use cases in which items don't change, like the sidebar items in this sample, or when the identity of an item isn't important. For all other use cases, or when in doubt as to which approach to use, populate diffable data sources and snapshots with proper identifiers.

See Also

Data

`{}` Implementing modern collection views

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

`{}` 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.