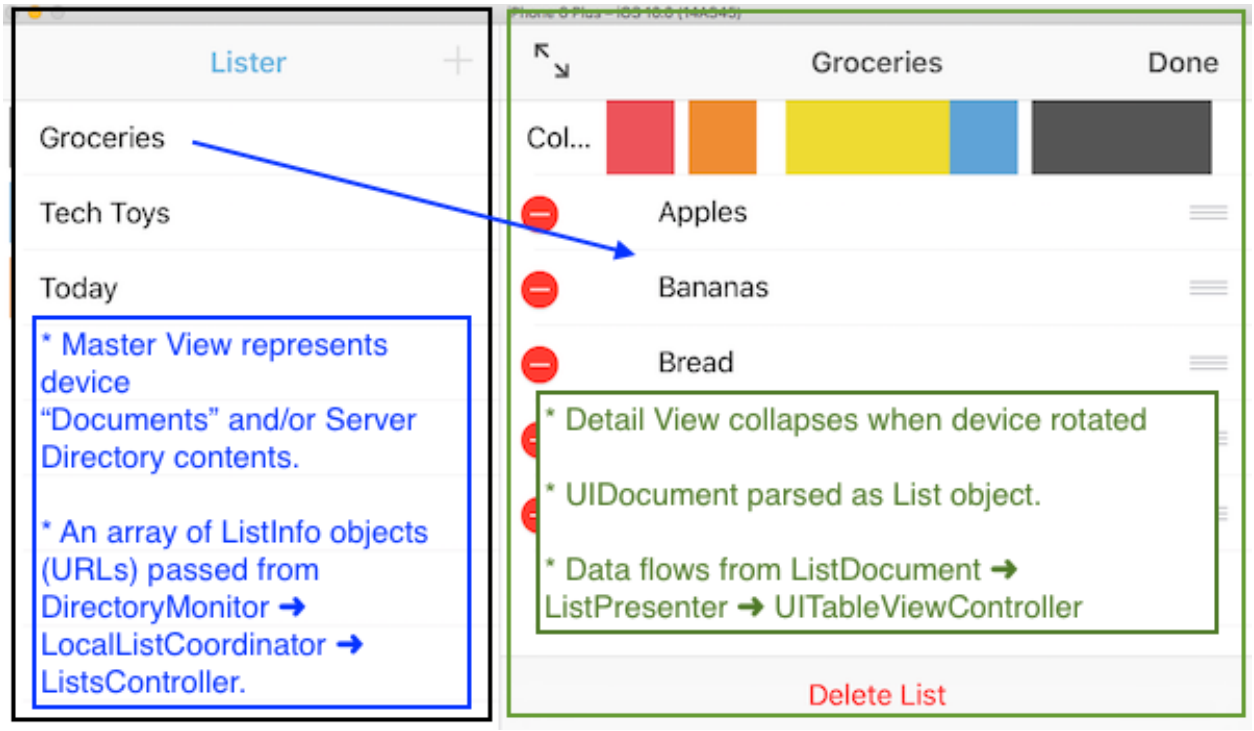
Mike LeRoy
KiteMike on GitHub
October 6, 2016

Apple Lister Example in Swift 3



The source code can be downloaded from:

https://github.com/kitemike/iosAppleListerExampleSwift3

Overview

The Lister app contains two functional units, as illustrated in the previous diagram. Each functional unit is expressed as a true

MVC design pattern. Traditional UITableVC Delegates blur the MVC distinctions.

The AppDelegate initiates the file monitoring process. File creation and deletion events are monitored by a DispatchSourceFileSystemObject. Events are posted to a DispatchQueue for processing. The DirectoryMonitor notifies its delegate (LocalListCoordinator) of changes.

The LocalListCoordinator class responds to processChangeToLocalDocumentsDirectory messages by comparing the previous and current directory file URLs. The ListsController responds to the ListCoordinatorDelegate.listCoordinatorDidUpdateContents messages by updating a list of ListInfo objects in the ListInfos array in the processContentChanges() method. URLs are

An array of URLs representing files are converted into ListInfo objects by the ListsController.

The ListDocumentsViewController displays ListInfo objects which allows the user to display the file contents.

translated into ListInfo objects.

The listInfoQueue is a DispatchQueue object. A listInfoQueue is a private, local queue to the `ListsController` that is used to perform updates.

The delegateQueue is an OperationQueue on which the `ListsController` object invokes delegate messages. The primary subscriber is the ListDocumentsViewController class, which displays file names and pushes the ListViewController onto the navigation stack. The interaction between the ListDocumentsViewController and ListsController classes is quite intricate. The set of ListInfo insert, update, move, delete and error handling operations are described in the ListsControllerDelegate protocol.

Relationship between ListsControllerDelegate protocols and ListDocumentsViewController for file management

The ListsControllerDelegate represents the model that drives file presentation. The example is a true Model-View-Controller. Ordinary UITableVC Delegate implementations blur the MVC distinctions. Of course, a disciplined MVC implementation is more complex, expensive and less direct. The more complex Lister implementation is necessary, because the file are not only external to the application, but also external to mobile devices.

The ListsController class is the Model in the MVC design pattern. The View is represented by the ListDocumentsViewController class. The LocalListCoordinator is the controller for device files. The CloudListCoordinator class was omitted from the original code to reduce the application to

minimal concepts. Restoring the CloudListCoordinator is a simple technical matter. I chose to push a UIWebView to allow the user to download example files in my own app. The main difference in the UIWebView implementation is restricting the implementation to the ListPresenterType. The AllListItemsPresenter class implements the ListPresenterDelegate, which is inappropriate for file downloads.

The ListPresenterType setList(), archiveableList, and presentedListItems stored properties are adequate for file downloading purposes. A reference to the ListsController is also necessary to append to the ListsController.ListInfos array. The setList() merges changes to reflect the directory file state.

The ListCoordinator.listCoordinatorDidUpdateContents() messages the ListsController to update the ListInfos array, which occurs in the ListsController.processContentChanges() method.

Actual ListsControllerDelegate Implementation

The conceptual explanation described in the previous paragraph is grouped into two ListDocumentsViewController extension to separate the code. The extensions allow you to copy/paste code snippets directly into your own app. The ListDocumentsViewController is much simpler than the ListViewController. First, the ListController is created by the AppDelegate and passed to ListDocumentsViewController. The AppDelegate also kicks off the Directory Monitoring process.

The operations are:

1. listsControllerWillChangeContent() - tableView.beginUpdates()
2. didInsertListInfo - tableView.insertRows()
3. didRemoveListInfo() - tableView.deleteRows()
4. didUpdateListInfo() - tableView.reloadRows()
5. listsControllerDidChangeContent() - tableView.endUpdates()

The ListDocumentsTVC logic is:

1. numberOfRowsInSection() - return listsController?.count
2. willDisplay() - let listInfo = listsController[indexPath.row]

Relationship between UIDocument, ListPresenter and ListViewController file open/parse/close

Tap on a ListDocumentsTVC cell to invoke the ListTVC, which displays the contents of a file, as a List. The ListTVC.configureWithListInfo() is called after creating the ListTVC and before ViewDidLoad() is invoked. ListTVC.configureWithListInfo() instantiates a ListDocument object. An AllListItemsPresenter (ListPresenterType) is passed to ListDocument.init(). The ListPresenterType.setList() manages the

file contents passed by the ListDocument.load() method. The ListDocument also contains a contents()/save() method. The ListDocument also manages undo/redo operations.

The ListPresenterDelegate.listPresenterDidRefreshCompleteLayout() results in ListTVC invoking tableView.reloadData(). Eventually, the configureListItemCell() is invoked,which obtains a List element from let listItem = listPresenter.presentedListItems[row - 1].

When the user deletes a cell from the ListTVC UI, AllListItemsPresenter removeListItem() is invoked. AllListItemsPresenter operations follow:

1. list.items.remove()
2. calls back to ListTVC
    - delegate?.listPresenterWillChangeListLayout()
        - tableView.beginUpdates()
    - delegate?.didRemoveListItem()
        - ListTVC invokes tableView.deleteRows()
    - delegate?.listPresenterDidChangeListLayout()
        - results in tableView.endUpdates()
3. self.document.updateChangeCount(), which eventually saves the document to the Documents directory.

When the user edits a row, the following calls are made:

1. listPresenter.updateListItem()
2. listPresenter.insertListItem()
3. triggerSave()

The ListPresenter responds with a series of callbacks to the ListTVC to make the appropriate tableView.xxx calls:

1. delegate?.listPresenterWillChangeListLayout()
2. delegate?.didUpdateListItem()
3. delegate?.listPresenterDidChangeListLayout()

The same pattern repeats itself when the user move a TVC cell. The canMoveListItem() utilizes the interesting "~=" operator.

Conclusion

The essential must-know implementations have been thoroughly explained. The next step is to take this new approach to heart. The benefit from the complexity is an MVC pattern in its truest form.

=== Explain ===

3. Difference between Table VC and traditional VCs, e.g., WebView to download a file

- Lightweight version of AllListItemsPresenter : ListPresenterType for read-only processing

4. How to parse JSON, rather than NSKeyedArchiver

5. Swift 2 to 3 conversion experience.

- Removed framework libraries
- 560 compiler error messages was tedious and time-consuming.
  - The single largest change is parameter naming.

    - Not providing all parameters results in functions not being invoked without any type of exception or halt in execution
- Migration tool from Swift 2 to 3 corrupted several files due to name collisions, i.e., NSURL to URL, when a property had URL as a name
- New DispathObject/GCD were complete rewrites, lacking Apple documentation
- Dynamic properties in XCode 8.0 Interface Builder crashes XCode. See CheckBox class.

7. Omitted code, e.g., iCloud, Watch, UserActivity to store state in UI Document between different devices,

8. Performance with nested arrays. I bounded my application to fixed size 16x16 arrays. The arrays involve calculations.

9. Multi-core GCD performance

10. My Split VC Delegate implementation, which corrects example code

11. The interesting Swift tricks