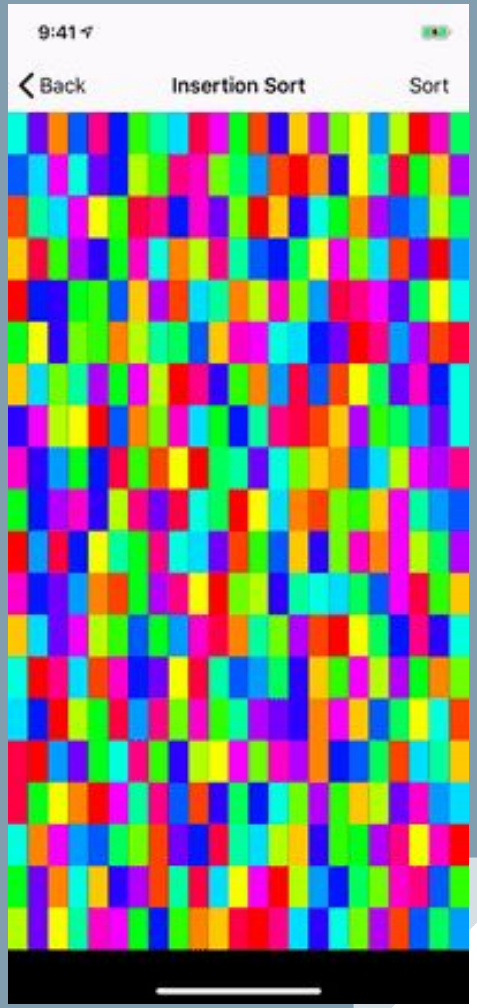
# IQListKit

Model driven
UITableView/UICollectionView

- What is **IQListKit**?
- What features it provides over traditional approach?

- **Model driven framework** which is used as a **delegate/dataSource** of **UITableView/UICollectionView**.
- So we **don’t have to implement** the UITableView / UICollectionView **delegate** and **dataSource** again and again.
- **Remove the need of buggy IndexPath**. Technically remove the possibilities of bugs and crashes.

# Minimum Requirements

- Xcode 11 and above
- iOS 9.0 and above
- Swift 5.0 and above

# Installation

## Cocoapods

Add `pod 'IQListKit'` to your podfile.

## Source Code

Drag and drop IQListKit directory from demo project to your project

## Swift Package Manager

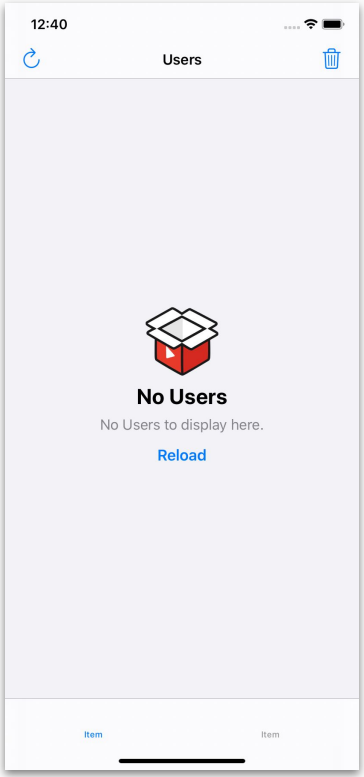
Follow the steps mentioned in README file of github repo

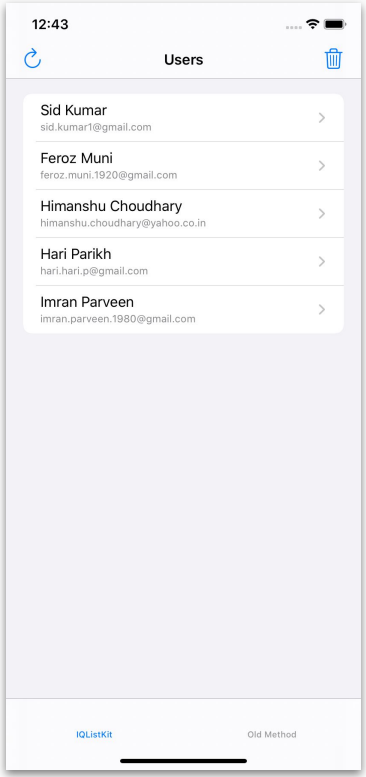
I’m already familiar with UITableView delegate/dataSource

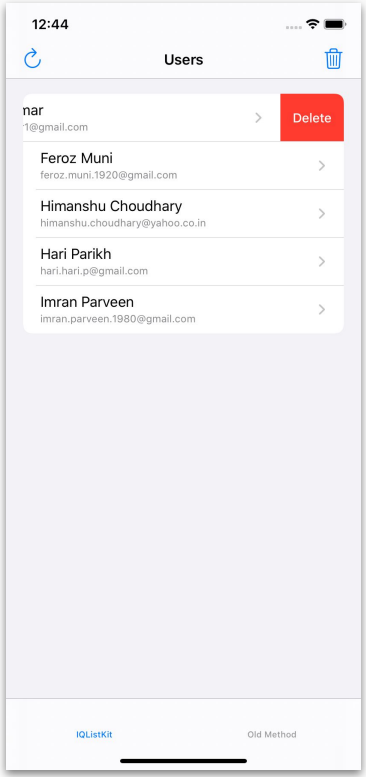
# Tell me how to use IQListKit?

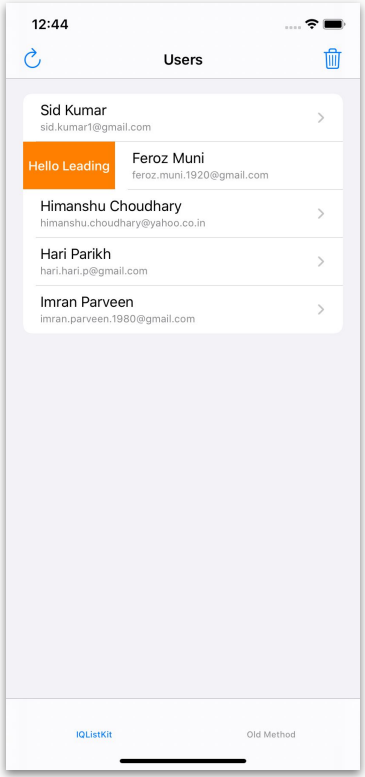
😀 No worries, we'll be learning using a simple example.

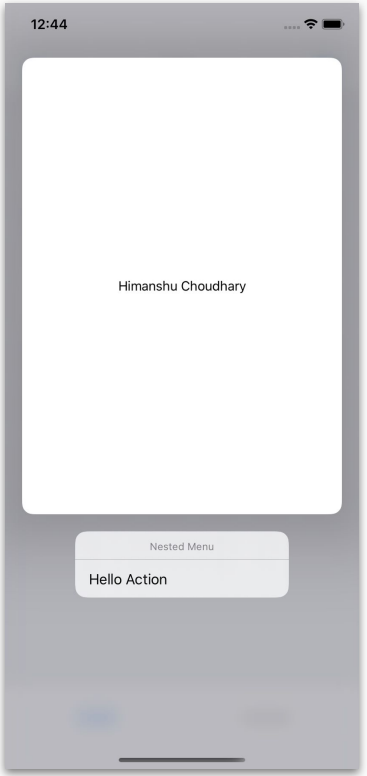
# Let's say we have to show a list of users



# Steps to implement IQListKit

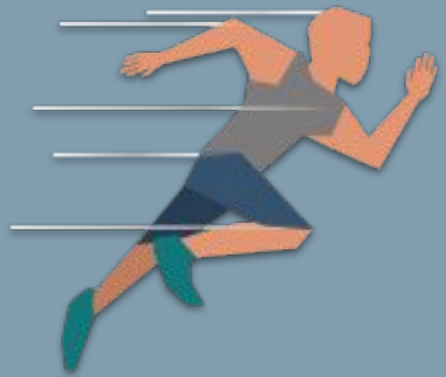
**In short (High Level)**

1. Modify our **User Model** to be **compatible** with **IQListKit**
2. Modify our **User Cell** to be **compatible** with **IQListKit**
3. Provide the **models** and **cell types** to **IQListKit**.

**Technically (Low Level)**

1. Confirm our **Model** to **Hashable**
2. Confirm our **Cell** to **IQModelableCell** protocol. Connect the **model** with the **cell**
3. Creating **IQList** object and configure it. Provide our **models** and **cell type** to **IQList**

# Let’s do a quick straight forward Implementation once



# User Model

```
//1. Conform to Hashable

struct User: Hashable {

    let id: Int          //user id

    let name: String     //user name

    let email: String    //user email
}
```

# UserCell

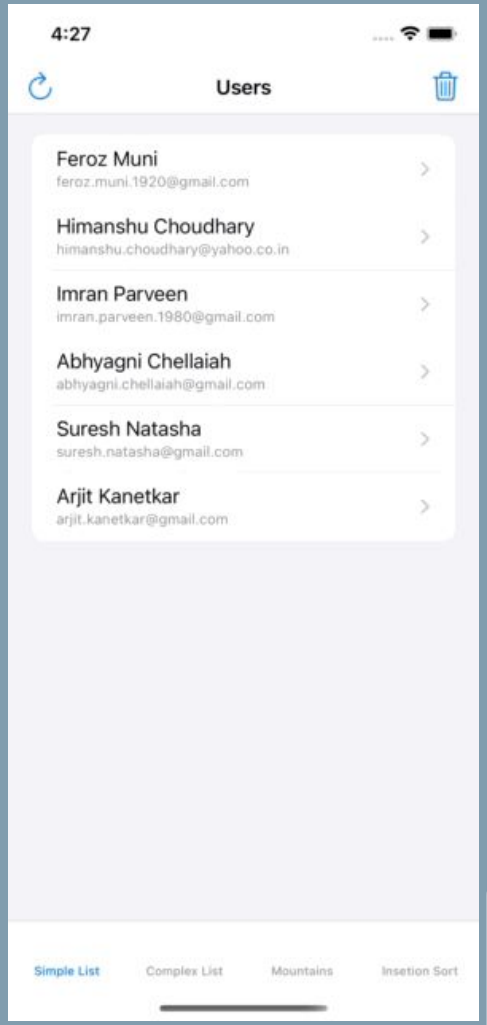
```
//1. Confirm IQModelableCell
//2. Expose a model property
//3. Implement didSet and connect model with the labels

class UserCell: UITableViewCell, IQModelableCell {

    @IBOutlet var labelName: UILabel!
    @IBOutlet var labelEmail: UILabel!

    var model: User? {
        didSet {
            guard let model = model else {
                return
            }

            labelName.text = model.name
            labelEmail.text = model.email
        }
    }
}
```

# Provide the models in performUpdates

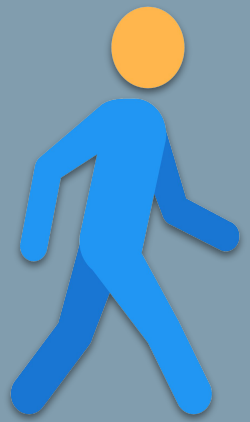
```
class UsersTableViewController: UITableViewController {
    private var users = [User]()
    private lazy var list = IQList(listView: tableView, delegateDataSource: self)

    func loadDataFromAPI() {
        APIClient.getUsersList({ [weak self] users in
            self?.users = users    //Updates the users array
            self?.refreshUI()      //Refresh the data
        }
    }

    func refreshUI() {
        list.performUpdates({ // We can think it like tableView.reloadData().

            let section = IQSection(identifier: "first")
            list.append(section)

            list.append(UserCell.self, models: users, section: section)
        })
    }
}

extension UsersTableViewController: IQListViewDelegateDataSource {
}
```

Data loaded without implementing **UITableViewDataSource**, **UITableViewDelegate.**

Specially without using any **IndexPath** calculations.



# Now let’s do it again in steps to understand the process



# Our User Model

```
struct User {

    let id: Int          //A unique id of each user
    let name: String     //Name of the user
    let email: String    //Email of the user
}
```

It is mandatory to make our User Model to conform to **Hashable** protocol. So before going ahear, we have to learn about the **Hashable** protocol.

# 😰 What is Hashable protocol? I never used it before.

- In short, hashable protocol is used to determine the **uniqueness of the object/variable**.
- Many types in the standard library already conform to Hashable:
  - `String`
  - `Int`
  - `Float`
  - `Double`
  - `Bool`
  - `Set`
- To **manually** confirm it we have to implement below functions

```
- func hash(into hasher: inout Hasher)
- static func == (lhs: Self, rhs: Self) -> Bool //Optional Equatable
```

# Step 1

## Confirm User Model to Hashable

### Method 1: Automatic Confirming

```
//We have Int and String variables in the struct
That's why we do not have to manually confirm the
hashable protocol.

struct User: Hashable {
    let id: Int
    let name: String
    let email: String
}

/*--------------*/
// let user1 = User(id: 1, name: "Arun", email: "")
// let user2 = User(id: 1, name: "Arun K", email: "")

// user1.hashValue == user2.hashValue //false, both
are unique
// user1 == user2 //false, not equal
```

### Method 2: Manually Confirming (Preferred)

```
struct User: Hashable {
    //Manually Confirming to the Hashable protocol
    func hash(into hasher: inout Hasher) {
        hasher.combine(id)
    }

    //Manually confirming to the Equatable protocol
    static func == (lhs: User, rhs: User) -> Bool {
        lhs.id == rhs.id && lhs.name == rhs.name &&
lhs.email == rhs.email
    }

    let id: Int
    let name: String
    let email: String
}
/*--------------*/
// let user1 = User(id: 1, name: "Arun", email: "")
// let user2 = User(id: 1, name: "Arun K", email: "")
// user1.hashValue == user2.hashValue //true, both same
// user1 == user2 //false, not equal
```

# Understand **IQModelableCell** protocol

Q. 🤨 What is **IQModelableCell** protocol? and how we should confirm it?

A. The **IQModelableCell** protocol says that, whoever adopts me, have to expose a variable named **model** and it can be **any type conforming to the Hashable**.

```
class MyCell: UITableViewCell, IQModelableCell {
    //...
     var model: AnythingHashableType?
}
```

# Step 2

## Confirm our **UserCell** to **IQModelableCell** protocol

```
//Let's say we have UserCell like this:

class UserCell: UITableViewCell {
    @IBOutlet var labelName: UILabel!
    @IBOutlet var labelEmail: UILabel!
}
```

# Confirm our **UserCell** to **IQModelableCell** protocol

## Method 1 & 2

### Method 1: Directly using our User model

```
class UserCell: UITableViewCell, IQModelableCell {

    @IBOutlet var labelName: UILabel!
    @IBOutlet var labelEmail: UILabel!

    //User model confirms the Hashable protocol
    var model: User?
}
```

### Method 2: typealias User model

```
class UserCell: UITableViewCell, IQModelableCell {

    @IBOutlet var labelName: UILabel!
    @IBOutlet var labelEmail: UILabel!

    //type aliasing the User model to a common name
    typealias Model = User

    //Model is a typealias of User
    var model: Model?
}
```

# Confirm our **UserCell** to **IQModelableCell** protocol

## Method 3

### Method 3: By creating a Hashable struct in each cell (Preferred)

```
class UserCell: UITableViewCell, IQModelableCell {

    @IBOutlet var labelName: UILabel!
    @IBOutlet var labelEmail: UILabel!

    struct Model: Hashable {
        let user: User
        let canShowMenu: Bool //custom parameter which can be controlled from ViewControllers
        let paramter2: Int  //Another customized parameter
        ... and so on (if needed)
    }

    //Our new Model struct confirms the Hashable protocol
    var model: Model?
}
```

# Step 3

## Connect the **model** with the **cell**

```
//To do this, we could easily do it by implementing the didSet of our model variable

class UserCell: UITableViewCell, IQModelableCell {

    @IBOutlet var labelName: UILabel!
    @IBOutlet var labelEmail: UILabel!

    var model: User? {  //For simplicity, we'll be using the 1st method
        didSet {
            guard let model = model else {
                return
            }

            labelName.text = model.name
            labelEmail.text = model.email
        }
    }
}
```

# Step 4

## Creating **IQList** object and configure it

```
//Let's say we have a UsersTableViewController like this:-

class UsersTableViewController: UITableViewController {

    private var users = [User]() //assuming the users array is loaded from somewhere e.g. API call response
    //...
    func loadDataFromAPI() {  //Get users list from API
        APIClient.getUsersList({ [weak self] result in
            switch result {
                case .success(let users):
                    self?.users = users   //Updates the users array
                    self?.refreshUI()     //Refresh the data (Will be implementing this later)
                case .failure(let error):
                    //Handle error
            }
        }
    }
}
```

# Step 4

## Creating **IQList** object and configure it

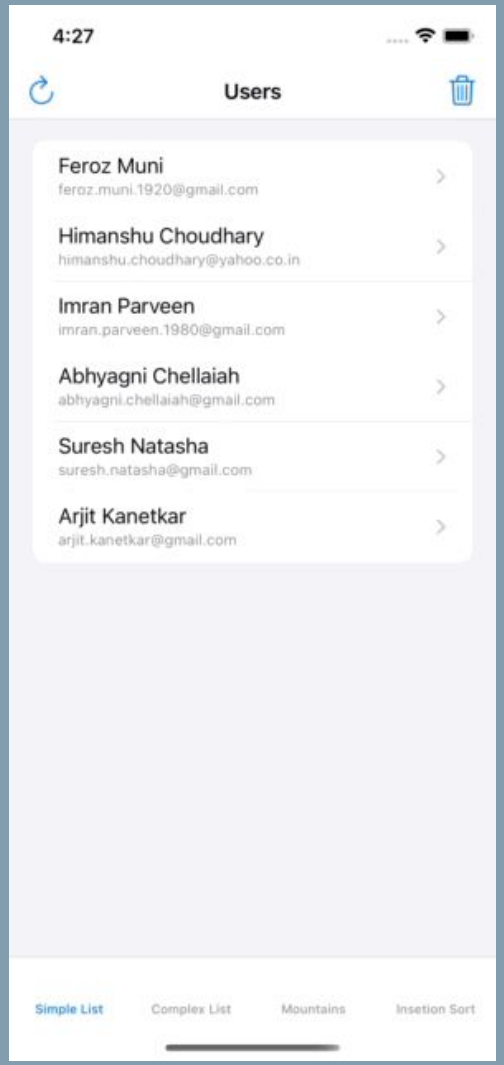
```
//Let's say we have a UsersTableViewController like this:-

class UsersTableViewController: UITableViewController {
    //...
    private lazy var list = IQList(listView: tableView, delegateDataSource: self)

    override func viewDidLoad() {
        super.viewDidLoad()
//        Optional configuration when there are no items to display
//        list.noItemImage = UIImage(named: "empty")
//        list.noItemTitle = "No User"
//        list.noItemMessage = "No users to display here."
//        list.noItemAction(title: "Reload", target: self, action: #selector(refresh(_:)))
    }
}

extension UsersTableViewController: IQListViewDelegateDataSource {
}
```

# **Step 5** Provide the models in performUpdates

```
class UsersTableViewController: UITableViewController {
    //...
    func refreshUI(animated: Bool = true) {
        list.performUpdates({ // We can think it like tableView.reloadData().

            let section = IQSection(identifier: "first")
            //let section = IQSection(identifier: 1)
            //let section = IQSection(identifier: "first", header: "I'm header", headerSize: CGSize(width:
view.width, height: 30), footer: "I'm footer", footerSize: CGSize(width: view.width, height: 50))
            list.append(section)

            list.append(UserCell.self, models: users, section: section) //If model created using Method 1 or 2
/*
            If model created using Method 3
            var models = [UserCell.Model]()
            for user in users {
                models.append(.init(user: user))
            }
            list.append(UserCell.self, models: models, section: section)
*/
        }, animatingDifferences: animated, completion: nil) //controls if changes should animate while reloading
    }
}
```

Data loaded without implementing **UITableViewDataSource**, **UITableViewDelegate.**

Specially without using any **IndexPath** calculations.



# I have a lot of questions like where is

- `tableView: cellForRowAt indexPath -> UITableViewCell`
- `tableView: didSelectRowAt indexPath`

 **Let us answer them one by one**

- `func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell`

**The IQListKit is a model-driven framework, so we'll be dealing with the Cell and models instead of the IndexPath. The IQListKit provides a couple of delegates to modify the cell before the cell display.**

```
extension UsersTableViewController: IQListViewDelegateDataSource {
    //WARNING: The indexPath of the display cell is provided for rare edge cases scenarious where we would like to know the position of the cell in tableView, don't use it until you extremely need it.
    func listView(_ listView: IQListView, modifyCell cell: IQListCell, at indexPath: IndexPath) {
        if let cell = cell as? UserCell { //Casting our cell as UserCell
            cell.delegate = self
            //Or additional work with the UserCell

            //🙂 Get the user object associated with the cell
            let user = cell.model

            //We discourage to use the indexPath variable to get the model object
            //😤 Don't do like this since we are model-driven list, not the indexPath driven list.
            //let user = users[indexPath.row]
        }
    }
}
```

- `func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)`

**Ahh, Don't worry about that. On cell selection we'll provide the user model associated with the cell directly 😍.**

```
extension UsersTableViewController: IQListViewDelegateDataSource {
      //WARNING: The indexPath of the display cell is provided for rare edge cases scenarious where we would like to know the position of the cell in tableView, don't use it until you extremely need it.
    func listView(_ listView: IQListView, didSelect item: IQItem, at indexPath: IndexPath) {

        if let model = item.model as? UserCell.Model { //😍 We get the user model associated with cell

            if let controller = UIStoryboard(name: "Main", bundle: nil).instantiateViewController(identifier: "UserDetailViewController") as? UserDetailViewController {

                controller.user = model //If used Method 1 or Method 2
                //  controller.user = model.user  //If used method 3
                self.navigationController?.pushViewController(controller, animated: true)
            }
        }
    }
}
```

- `func tableView(_ tableView: UITableView, estimatedHeightForRowAt indexPath: IndexPath) -> CGFloat`
- `func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat`

**Because these methods mostly return values based on cell and it's model, we have moved these configurations to cell and is part of IQCellSizeProvider protocol. Default behaviour can be overrided.**

```
class UserCell: UITableViewCell, IQModelableCell {
    //...
    static func estimatedSize(for model: AnyHashable?, listView: IQListView) -> CGSize {
        return CGSize(width: listView.frame.width, height: 100)
    }

    static func size(for model: AnyHashable?, listView: IQListView) -> CGSize {
        if let model = model as? Model {
            var height: CGFloat = 100
            // return height based on the model
            return CGSize(width: listView.frame.width, height: height)
        }
        return CGSize(width: listView.frame.width, height: 100) //Or return a constant height
//        return CGSize(width: listView.frame.width, height: UITableView.automaticDimension) //Or UITableView.automaticDimension for dynamic behaviour
    }
}
```

- `func tableView(_ tableView: UITableView, leadingSwipeActionsConfigurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration?`
- `func tableView(_ tableView: UITableView, trailingSwipeActionsConfigurationForRowAt indexPath: IndexPath) -> UISwipeActionsConfiguration?`
- `func tableView(_ tableView: UITableView, editActionsForRowAt indexPath: IndexPath) -> [UITableViewRowAction]?`

**These are part of IQCellActionsProvider protocol because they also depend on the cell and it's model.**

```
class UserCell: UITableViewCell, IQModelableCell {

    func trailingSwipeActions() -> [IQContextualAction]? {
        let action = IQContextualAction(style: .normal, title: "Hello Trailing") { [weak self] (action,
completionHandler) in
            completionHandler(true)
            guard let self = self, let user = self.model else {
                return
            }
            //Do your stuffs here
        }
        //action.image = UIImage(named: "delete")
        action.backgroundColor = UIColor.purple
        return [action]
    }

    @available(iOS 11.0, *)
    func leadingSwipeActions() -> [IQContextualAction]? {    return nil  }
}
```

# Other useful delegate methods

```
extension UsersTableViewController: IQListViewDelegateDataSource {

    //...

    //Cell will about to display
    func listView(_ listView: IQListView, willDisplay cell: IQListCell, at indexPath: IndexPath) {
        //Do your stuffs here
    }

    //Cell did end displaying
    func listView(_ listView: IQListView, didEndDisplaying cell: IQListCell, at indexPath: IndexPath) {
        //Do your stuffs here
    }
}
```

# Other useful data source methods

```
extension UsersTableViewController: IQListViewDelegateDataSource {

    //...

    //Return the size of an Item, for tableView the size.height will only be effective
    func listView(_ listView: IQListView, size item: IQItem, at indexPath: IndexPath) -> CGSize? {
        //Calculate the size or height of Cell
        return CGSize(width: listView.frame.width, height: heightOfCell)
    }

    //Return the headerView of section
    func listView(_ listView: IQListView, headerFor section: IQSection, at sectionIndex: Int) -> UIView? {
        //Create the headerView for section
        return headerView
    }

    //Return the footerView of section
    func listView(_ listView: IQListView, footerFor section: IQSection, at sectionIndex: Int) -> UIView? {
        //Create the footerView for section
        return footerView
    }
}
```

# Other useful IQModelableCell properties

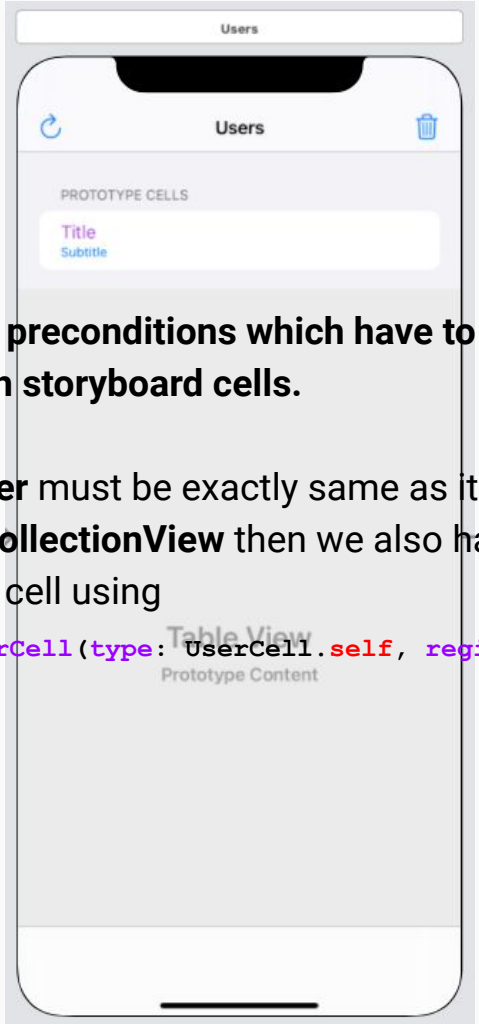
```
class UserCell: UITableViewCell, IQModelableCell {

    //...

    var isHighlightable: Bool { //IQSelectableCell protocol
        return true
    }

    var isSelectable: Bool {    //IQSelectableCell protocol
        return false
    }
}
```

# Other useful IQModelableCell methods

```
class UserCell: UITableViewCell, IQModelableCell {

    //...

    // contextMenu configuration of the cell
    @available(iOS 13.0, *)
    func contextMenuConfiguration() -> UIContextMenuConfiguration? {
        return nil
    }

    // contextMenu menu customized preview view (If different)
    @available(iOS 13.0, *)
    func contextMenuPreviewView(configuration: UIContextMenuConfiguration) -> UIView? {
        return viewToBePreview
    }

    // Context menu preview is tapped, now you probably need to show the preview controller
    @available(iOS 13.0, *)
    func performPreviewAction(configuration: UIContextMenuConfiguration, animator:
UIContextMenuInteractionCommitAnimating) {
        //Show previewViewController
    }
}
```

# Known issues and Workarounds

# IQListKit! 😡 Why are you not loading my cell created in storyboard?




**There are certain preconditions which have to be satisfied in order to work with storyboard cells.**

1. Cell **identifier** must be exactly same as its class name.
2. If using **UICollectionView** then we also have to manually register our cell using

```
list.registerCell(type: UserCell.self, registerType: .storyboard)
```

# I have a large data set around 10,000 records and `list.performUpdates` method takes time to animate changes 😟. What can I do?

You will not believe that **performUpdates** method is **Background Thread Safe** 😍. We can call it in background and can show a loading indicator until the changes are calculated in background and rendered in main thread.

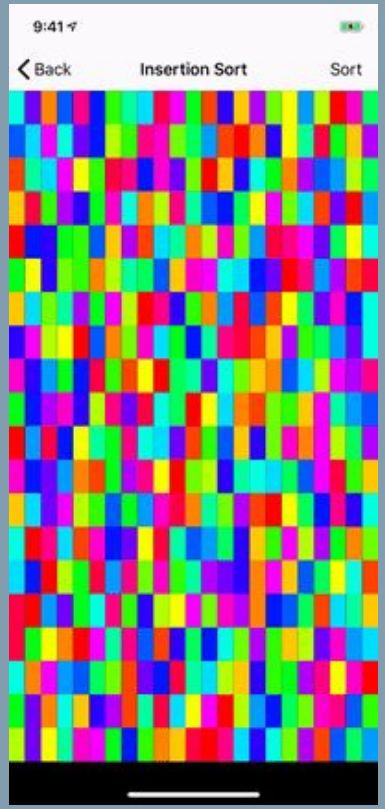
```
loadingIndicator.startAnimating() //Show loading indicator

DispatchQueue.global().async {     //Perform updates in background

    self.list.performUpdates({

        let section = IQSection(identifier: "first")
        self.list.append(section)
        self.list.append(UserCell.self, models: users, section: section)

    }, animatingDifferences: animated, completion: {
        self.loadingIndicator.stopAnimating() //Hide indicator, completion is called in main thread
    })
}
```

# Special Thanks to

**Apple** for **NSDiffableDataSourceSnapshot** (iOS 13 and above)
https://developer.apple.com/documentation/uikit/nsdiffabledatasourcesnapshot

**Ryo Aoyama** for **DiffableDataSources** (iOS 12 and below)
https://github.com/ra1028/DiffableDataSources

# Download the demo project to see it in action





Questions, suggestions and improvements can be contributed through github issues

https://github.com/hackiftekhar/IQListKit