GeekBrains

Пользовательский интерфейс iOS-приложений

Кастомизация коллекций

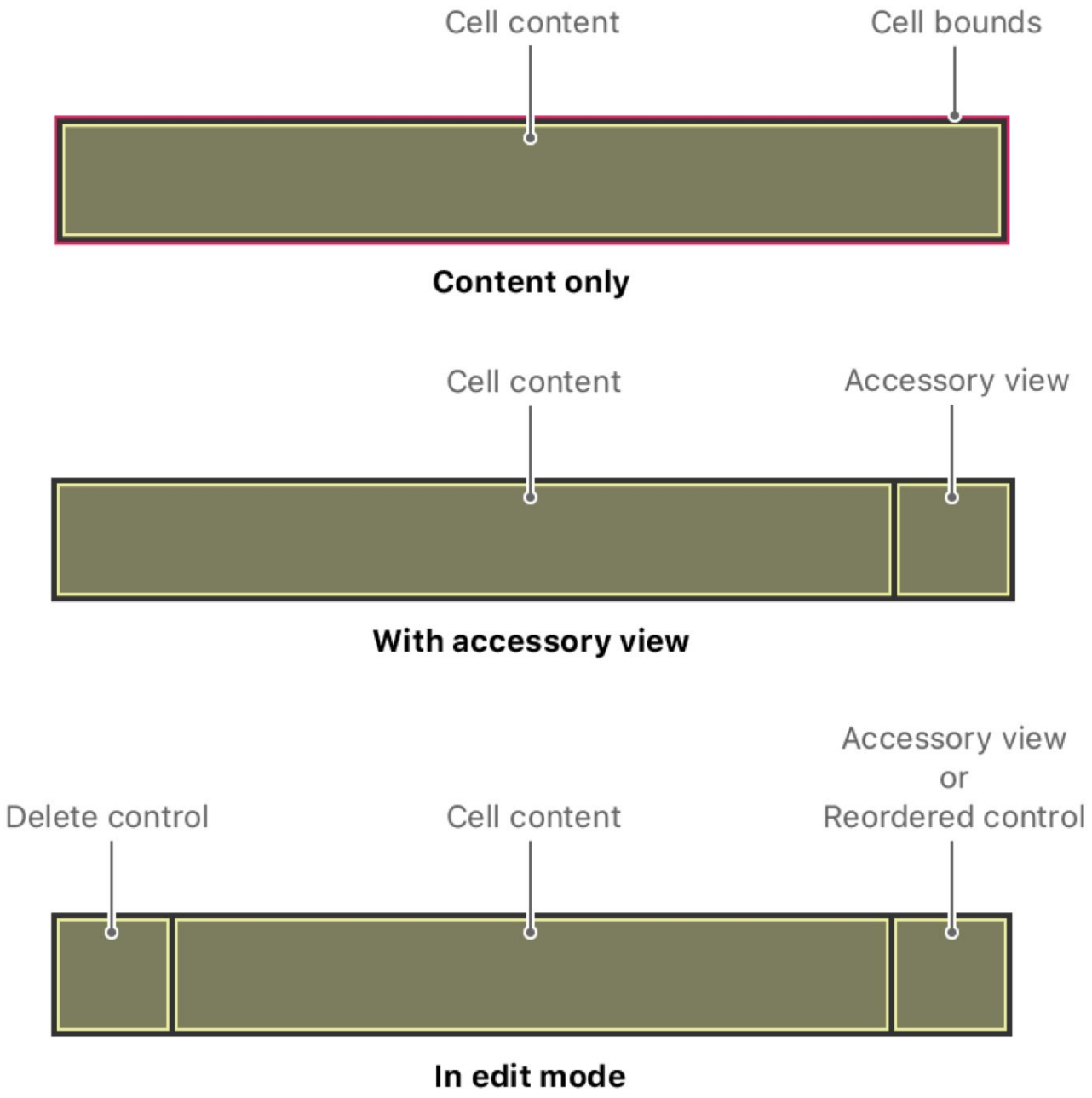
Создание кастомных ячеек таблицы, header и footer view, кастомных ячеек коллекции, supplementary views. Работа с UICollectionViewFlowLayout. Создание кастомного collection view layout.

Оглавление

Кастомизация UITableView

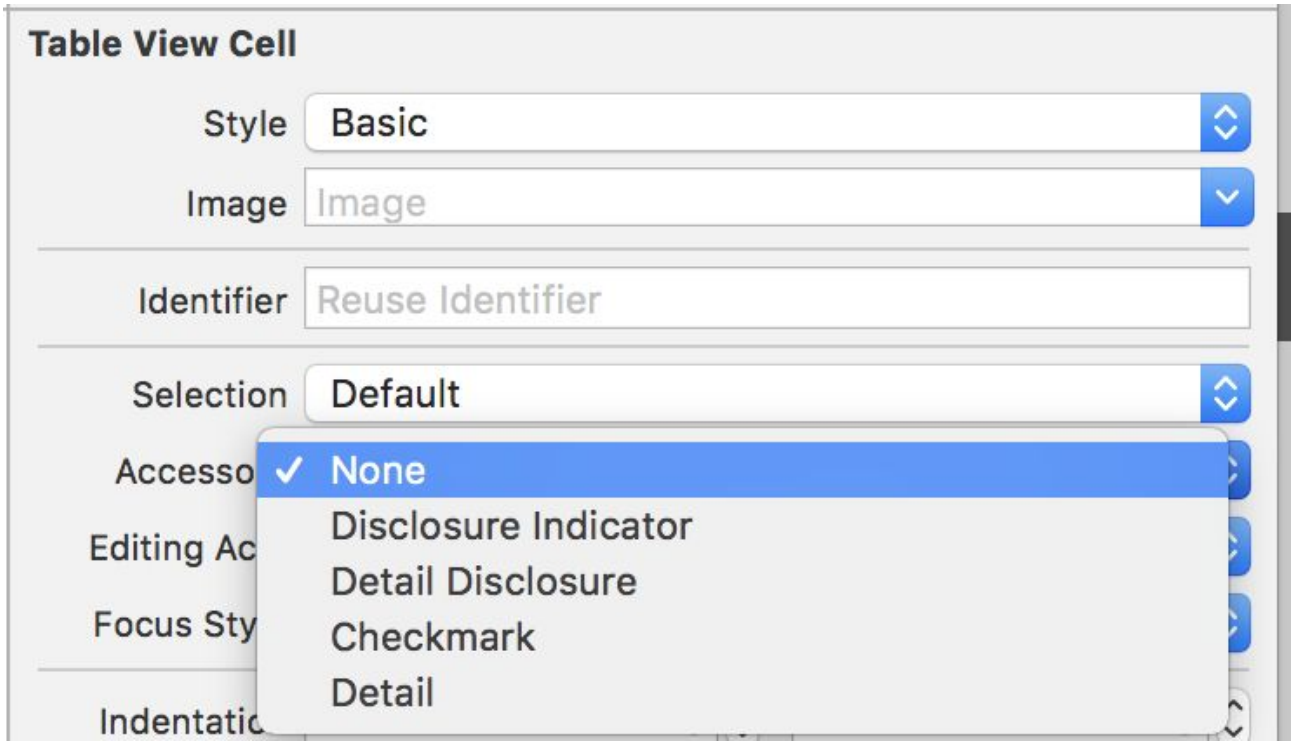
Анатомия ячеек таблицы

Внутри каждой ячейки таблицы есть несколько контейнеров для разных целей. Схематически они выглядят так:



Основной контейнер — это **ContentView**. Он содержит весь контент — лейблы, картинки и прочие элементы. В **ContentView** следует добавлять свои UI-элементы.

Accessory view содержит вспомогательный компонент — индикатор, переключатель. Можно установить один из стандартных **accessory view**:



Также возможно установить собственный **accessory view**. Для этого нужно применить свойство **accessoryView** у ячейки.

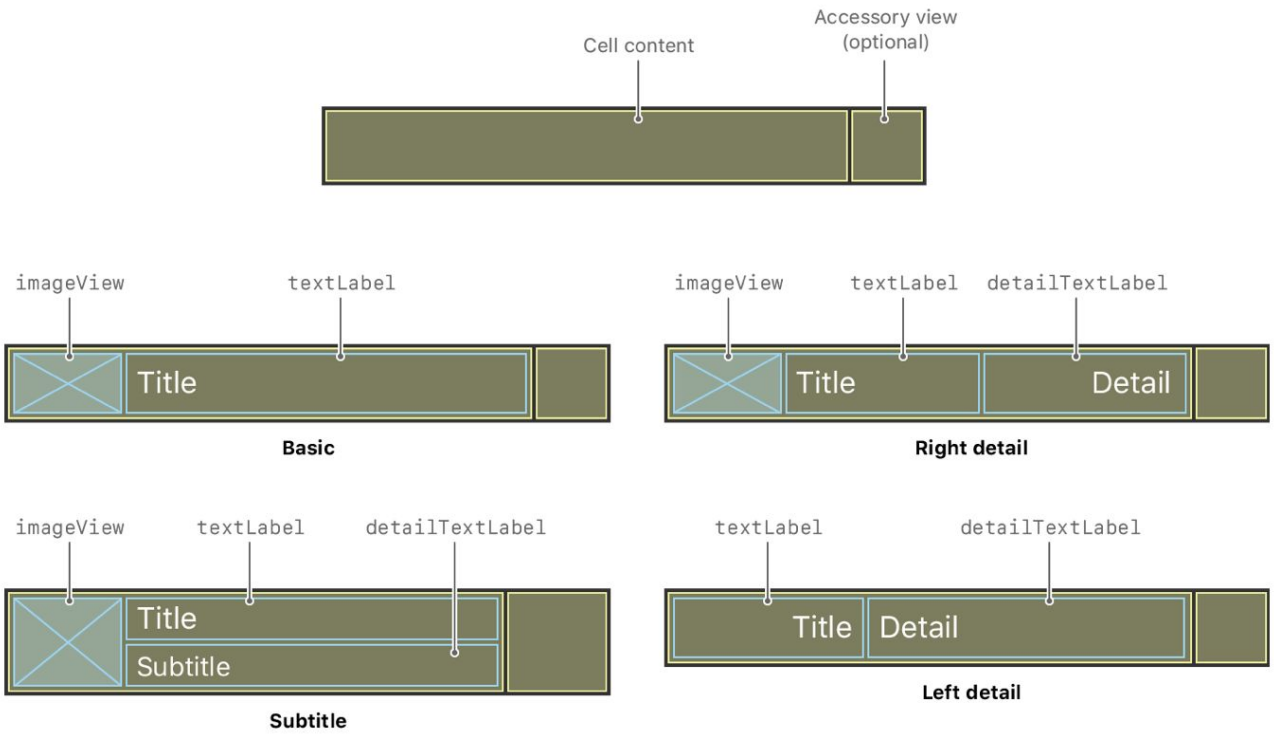
Контейнер **delete control** — это компонент, который появляется при активации режима редактирования.

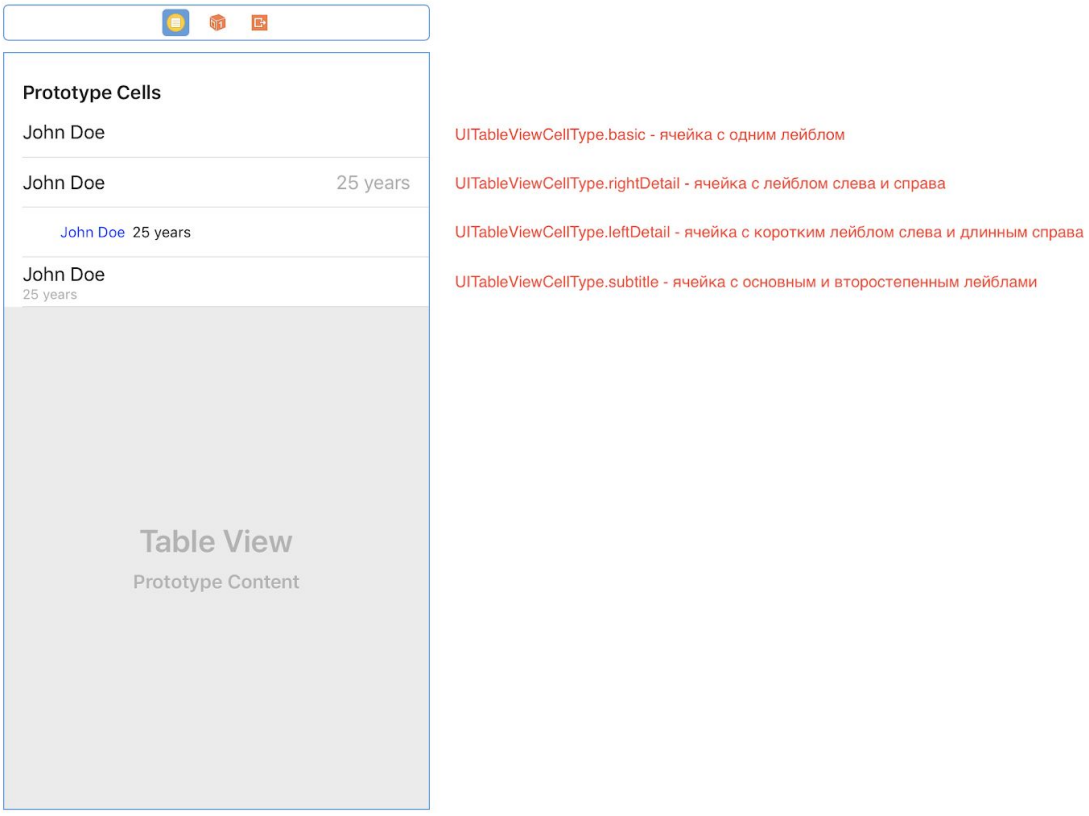
Переиспользование ячеек

UITableView реализует механизм переиспользования ячеек — их создается несколько, чтобы отобразить в видимой области экрана (и небольшой запас). Представьте, что у вас на экране по высоте возможно разместить максимум 10 ячеек с друзьями пользователя, а таблица должна отобразить всех — например, 1000. При скролле ячейка, которая пропадает из вида, конфигурируется заново и отображается в противоположной части таблицы — таким образом не происходят затратные процессы постоянного создания и удаления ячеек в памяти, да и ресурсов памяти требуется значительно меньше. Важно помнить: раз ячейка из памяти не удаляется, то и все ее **subview** сохраняются с установленными значениями. То есть **UILabel** хранит установленный ранее текст и форматирование. **UIImageView** — отображаемую фотографию и свойства ее отображения, **UISlider** — выбранное значение слайдера. Так что при переиспользовании ячеек необходимо переопределить свойства их элементов на новые или сбросить на значения по умолчанию.

Создание кастомных ячеек

По умолчанию фреймворк **UIKit** предоставляет несколько видов ячеек, которые определены в перечислении **UITableViewCellType**:



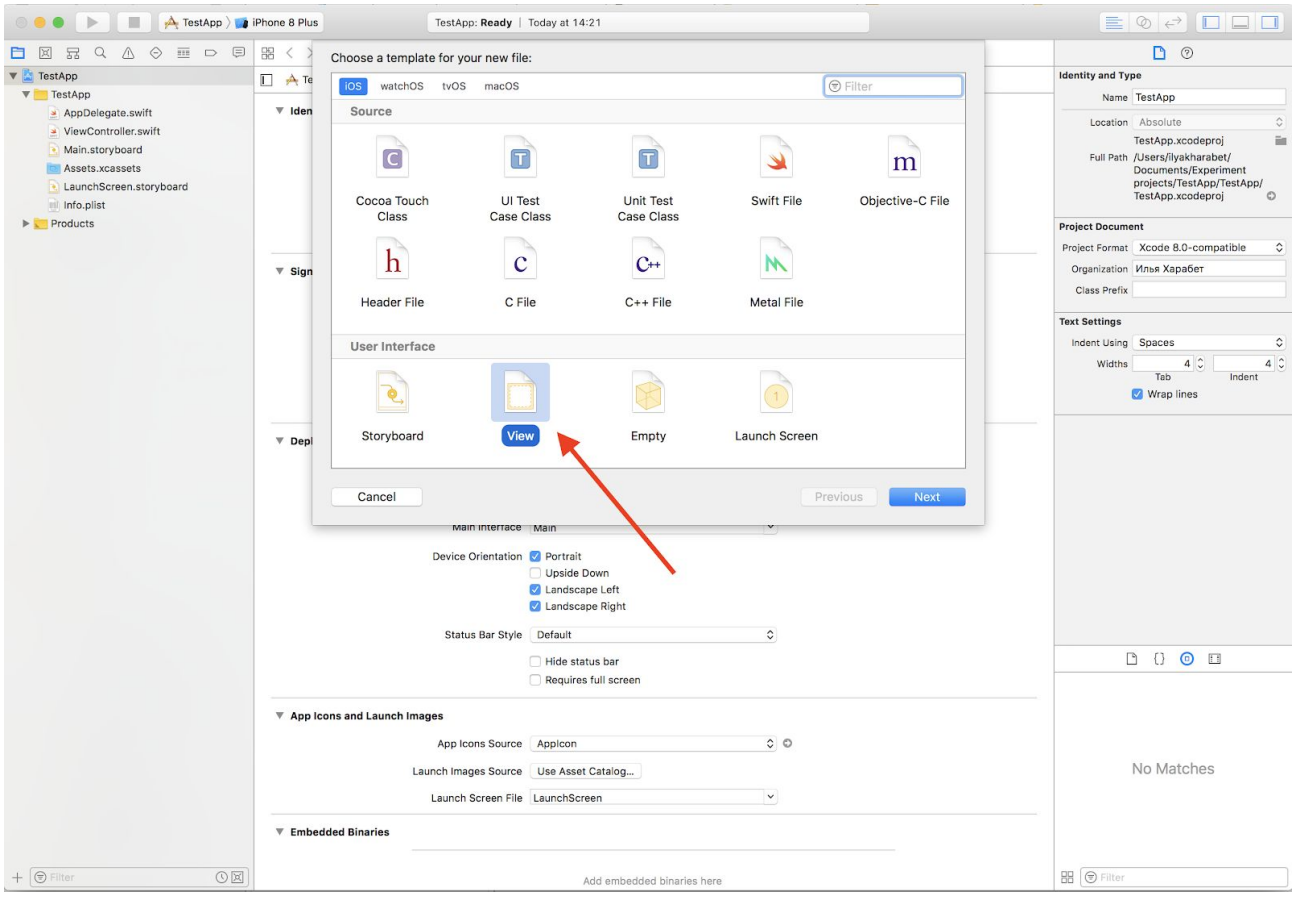


Эти типы ячеек используются во многих приложениях от Apple, но редко — от других компаний, так как они не соответствуют разработанному дизайну приложения. Для более тонкой настройки можно создать свою кастомную ячейку. Для этого необходимо добавить наследника класса **UITableViewCell**. В этом классе можно объявить **IBOutlet's** для вложенных UI-компонентов и нужно переопределить метод **prepareForReuse**. Он вызывается в момент, когда ячейка будет переиспользована. В нем

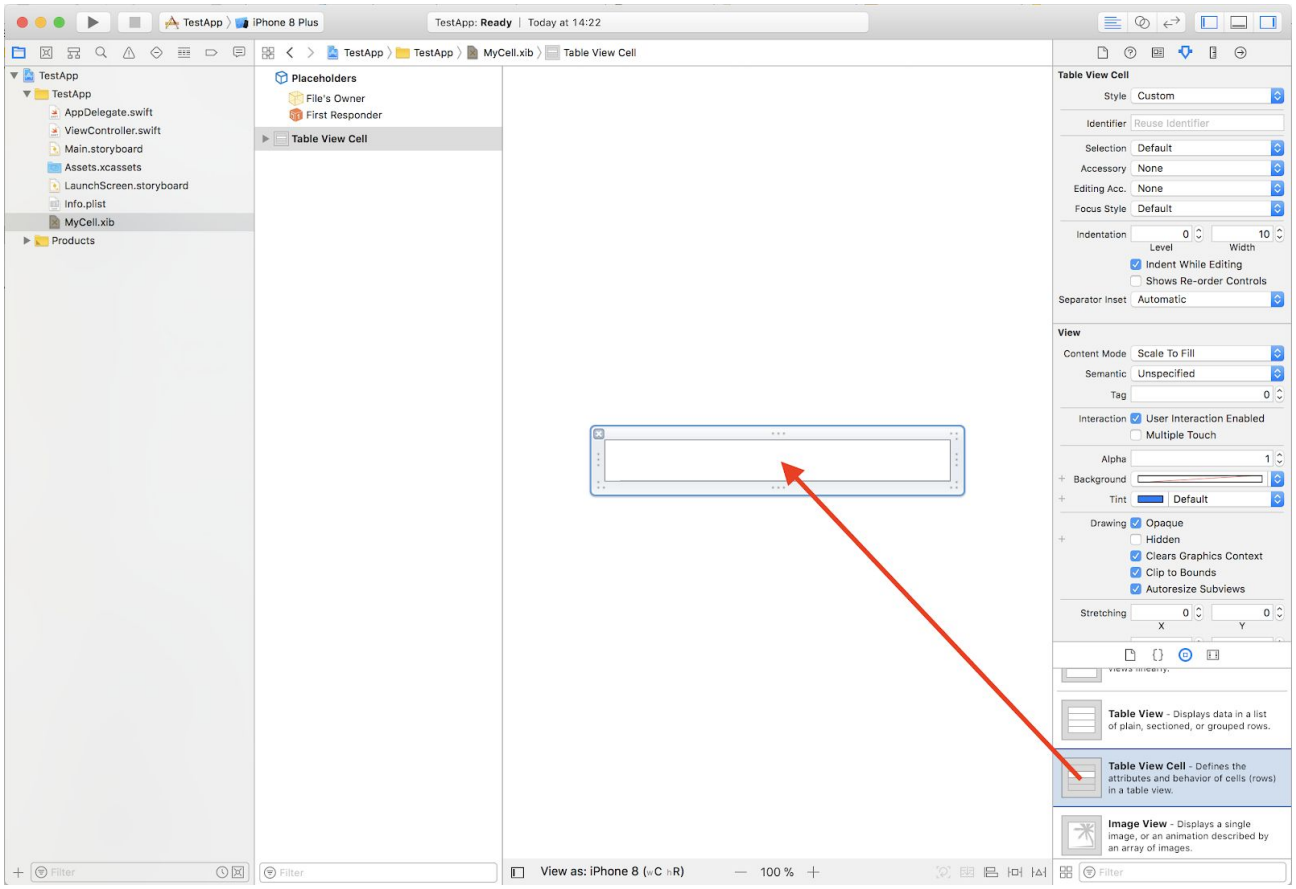
необходимо подготовить ячейку — убрать надписи с лейблов и удалить картинки из **UIImageView**. Если этого не сделать, в ячейке могут остаться данные из предыдущего состояния.

Для этой ячейки нужно сделать xib-файл или создать ее в таблице, которая будет ее использовать. Рассмотрим добавление ячейки в xib-файле.

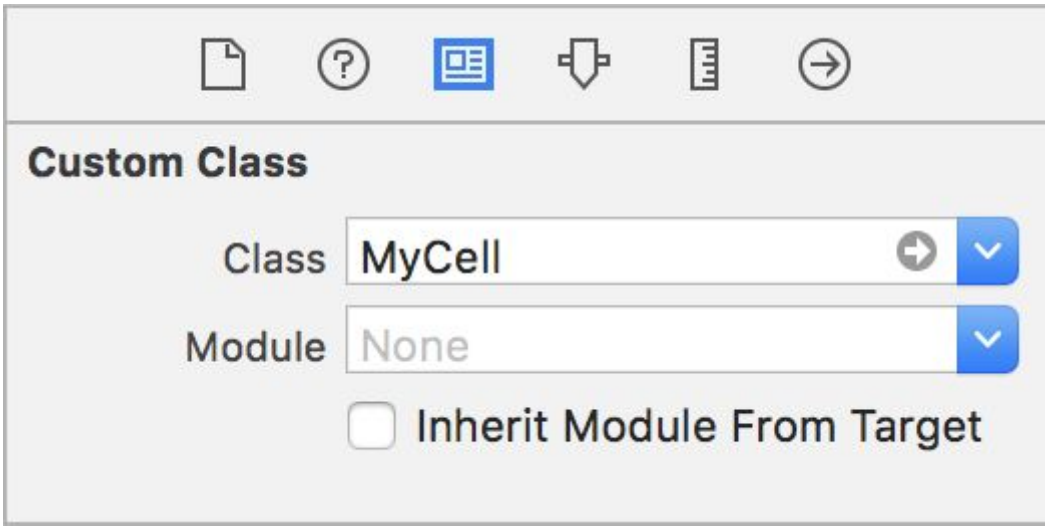
Создадим xib-файл:



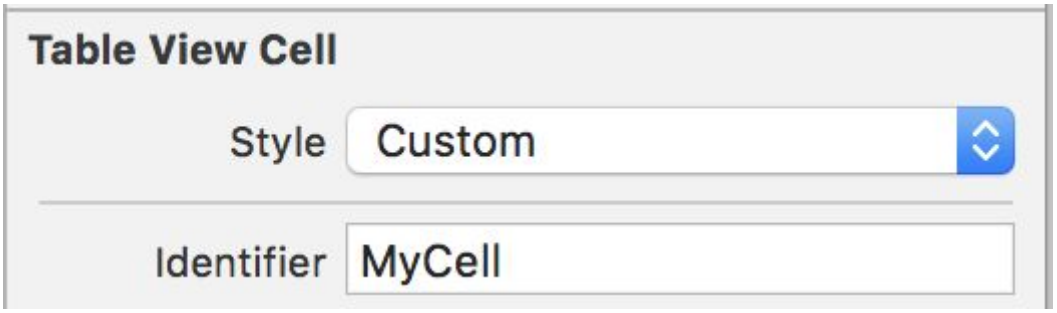
Удалим тот **view**, который добавился автоматически, и поставим **UITableViewCell** из списка UI-компонентов:



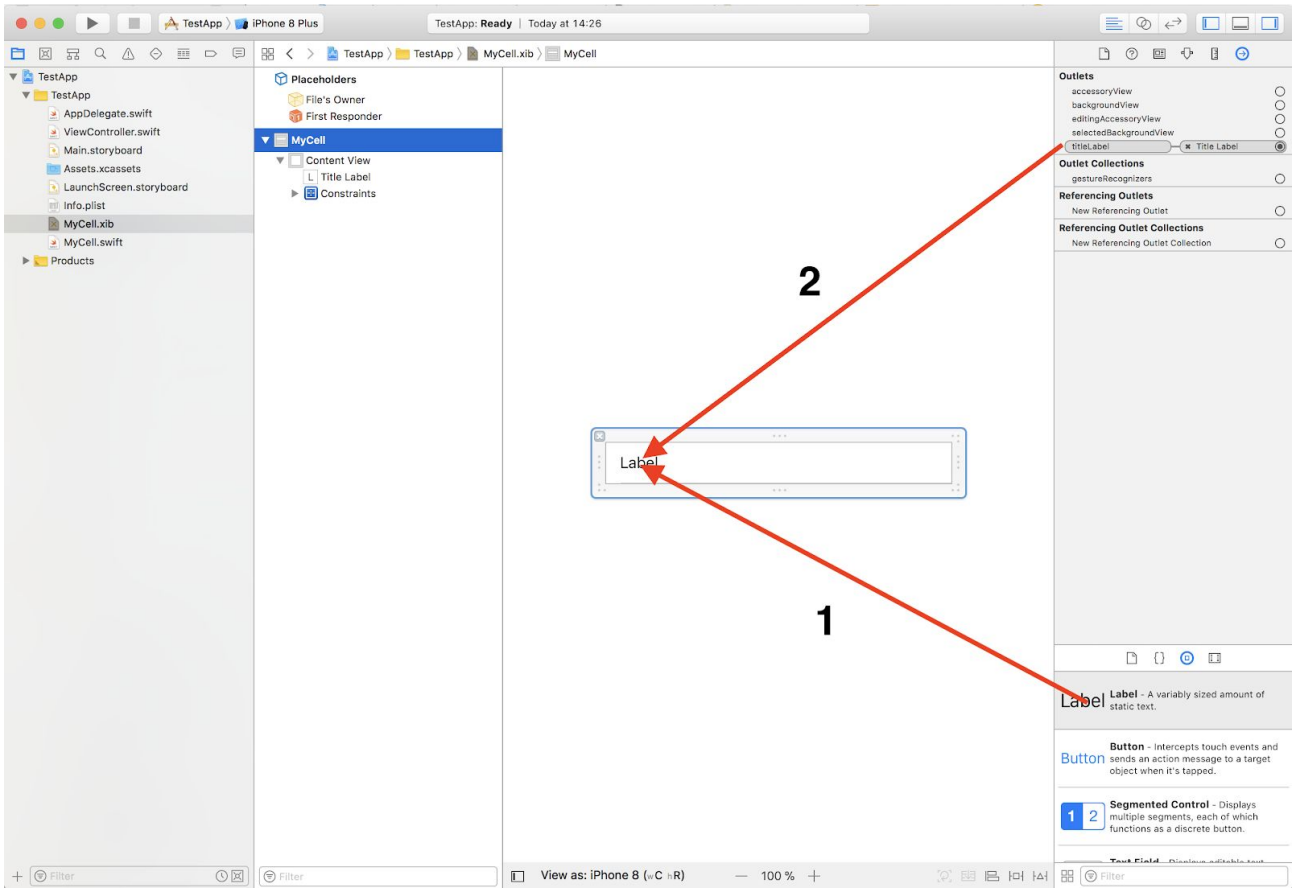
Затем установим этой ячейке нужный класс:



Установим **reuseIdentifier**, чтобы ячейку можно было идентифицировать впоследствии:



Добавим необходимые компоненты, расставим констрейнты и соединим компоненты с **IBOutlet** класса:

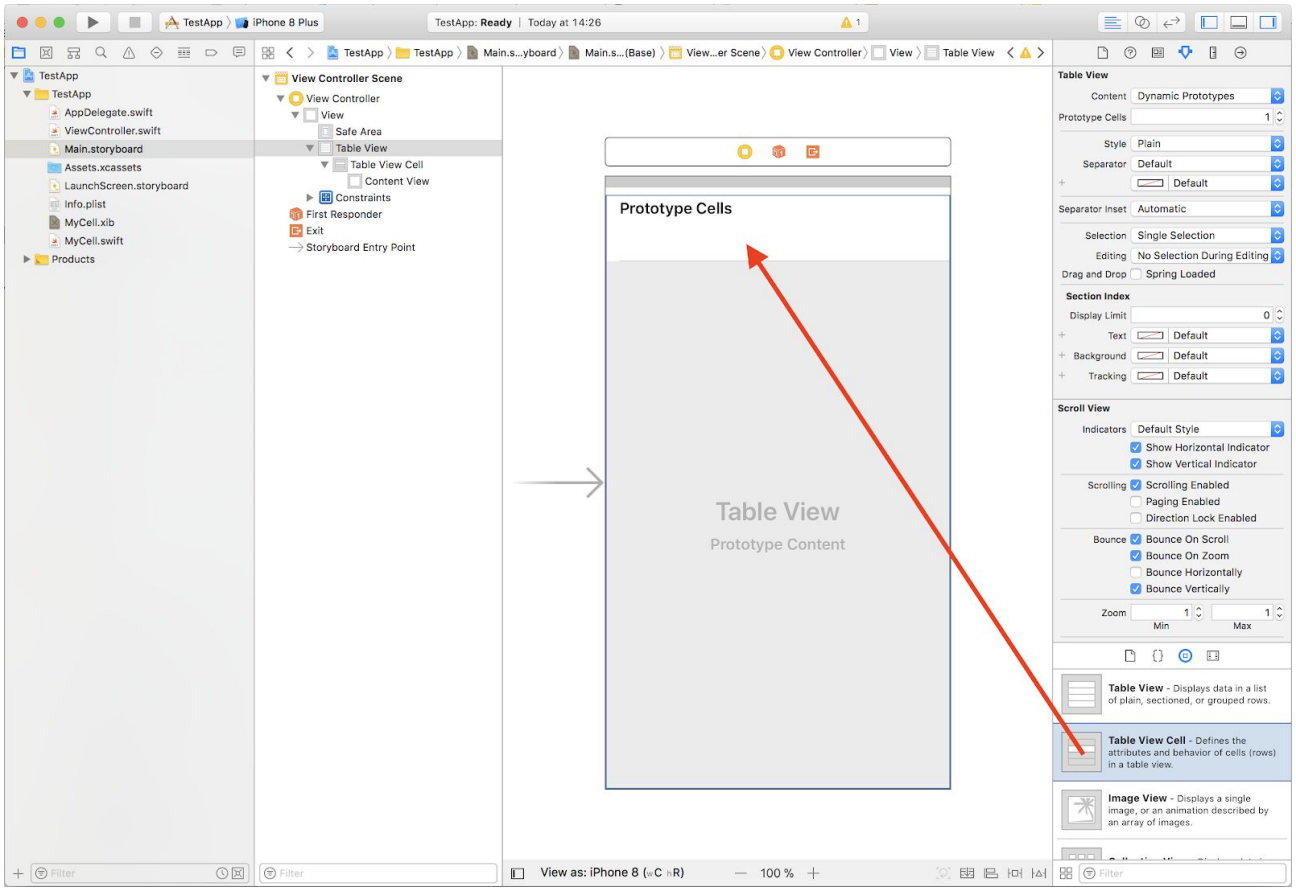


Ячейку нужно зарегистрировать в таблице, чтобы она отображалась:

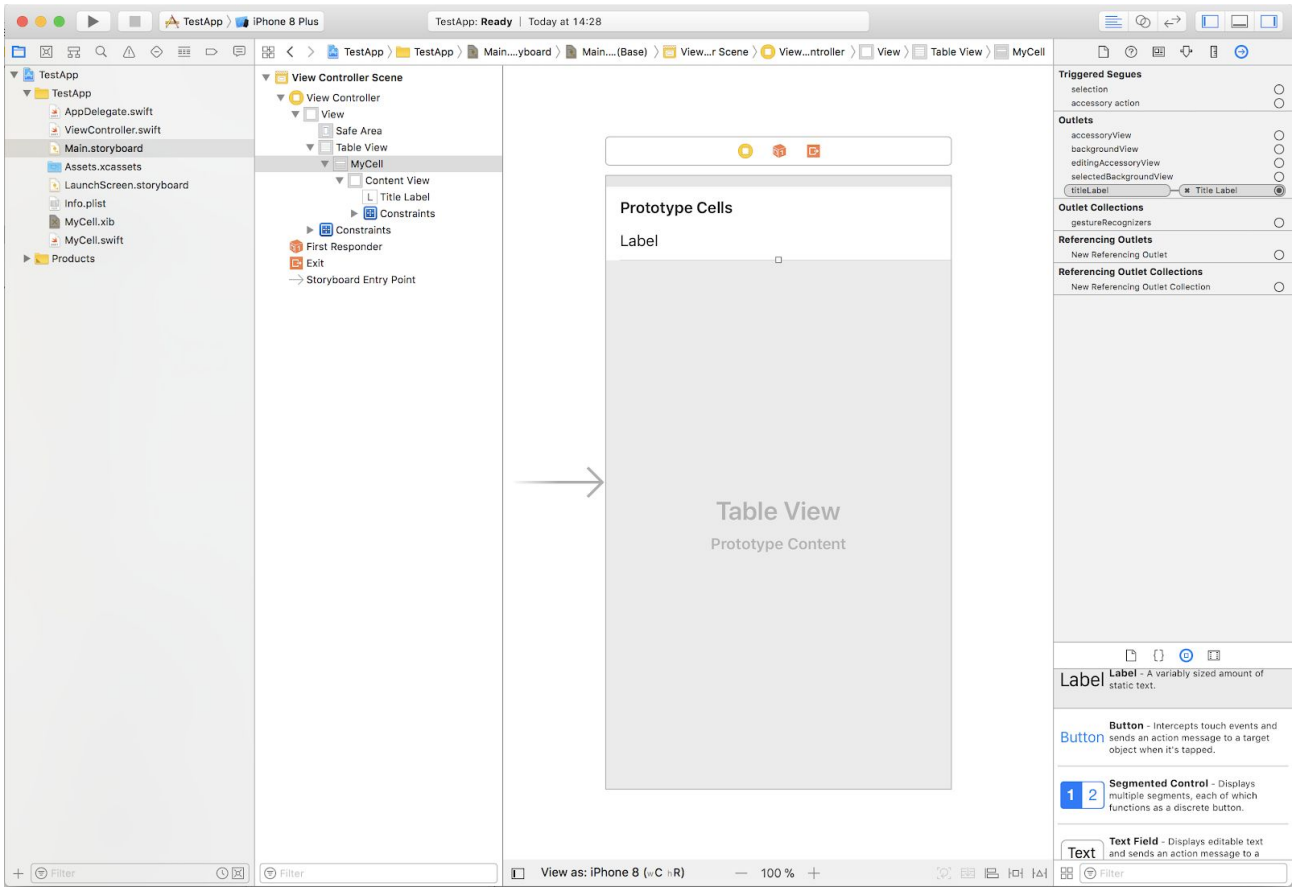
```
tableView.register(UINib(nibName: "MyCell", bundle: nil),
                   forCellReuseIdentifier: "MyCellId")
```

В качестве первого аргумента нужно передать xib-файл, а второго — тот **reuseIdentifier**, который установили в xib-файле для этой ячейки.

Второй способ гораздо проще. Нужно добавить прототип ячейки в таблицу:



После этого устанавливаем класс ячейки и **reuseIdentifier** и добавляем компоненты, как в случае с **xib**:



Ячейку, которая была создана таким образом, не нужно регистрировать в таблице — это произойдет автоматически.

Чтобы использовать созданную ячейку, ее нужно запросить у таблицы следующим образом:

```
func tableView(_ tableView: UITableView,
               cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "MyCellId",
                                                 for: indexPath) as! MyCell
        // Конфигурируем ячейку
        return cell
    }
```

Метод **dequeueReusableCell** вернет ячейку по переданному **reuseIdentifier** для заданного индекса. После этого можно сконфигурировать ячейку.

Создание кастомных header и footer для секций таблицы

У каждой секции таблицы есть **header** и **footer** — специальные ячейки, которые находятся перед ячейками секции и после них соответственно. Для них тоже применяется механизм переиспользования, и их нужно регистрировать и получать с помощью метода **dequeueReusableHeaderFooterView**.

Чтобы добавить свой **header** или **footer**, нужно создать наследника класса **UITableViewHeaderFooterView**. После этого, по аналогии с ячейкой, нужно создать **IBOutlet's**, **xib** или прототип в таблице и переопределить метод **prepareForReuse**.

Создание кастомных header и footer таблицы

У таблицы, как и у секций, есть **header** и **footer**. Они располагаются вверху и внизу таблицы. Не переиспользуются и чаще всего отображают статичный контент — например, строку поиска или индикатор загрузки. Чтобы добавить в таблицу header или **footer**, нужно установить свойства **tableHeaderView** и **tableFooterView**. В роли этих **view** может выступать любой наследник **UIView**. Важно установить **frame** у **view** перед тем, как добавлять ее в качестве **header** или **footer**. Чтобы изменить размер **header** или **footer**, нужно установить новый **frame** и вызвать метод **reloadData** у таблицы.

Динамическое добавление, удаление и обновление ячеек

Чаще всего для обновления таблицы используется метод **reloadData**. Но чтобы добавить, удалить или обновить определенную ячейку или даже секцию, существуют специальные методы:

- **reloadRows(at indexPaths: [IndexPath], with animation: UITableViewRowAnimation)** — обновляет ячейки, находящиеся по переданным индексам;
- **insertRows(at indexPaths: [IndexPath], with animation: UITableViewRowAnimation)** — вставляет ячейки по переданным индексам;
- **deleteRows(at indexPaths: [IndexPath], with animation: UITableViewRowAnimation)** — удаляет ячейки по переданным индексам;
- **reloadSections(section: IndexSet, with animation: UITableViewRowAnimation)** — обновляет ячейки по переданным индексам;
- **insertSections(sections: IndexSet, with animation: UITableViewRowAnimation)** — вставляет секции по переданным индексам;
- **deleteSections(sections: IndexSet, with animation: UITableViewRowAnimation)** — удаляет ячейки по переданным индексам.

В каждый из этих методов передается параметр **animation**. Это тип анимации, с которым будет происходить действие. Основные виды анимаций:

- **none** — действие произойдет без анимации;
- **automatic** — анимация будет подобрана автоматически;
- **fade** — ячейка будет постепенно менять прозрачность;
- **right** — ячейка будет появляться справа (или удаляться вправо).

Если вызвать несколько этих методов один за другим, действия будут происходить по очереди. Чтобы сделать несколько действий одновременно, нужно выполнить их в блоке обновлений. Для этого сначала запускают обновления, вызывают несколько методов и завершают обновления. Это будет выглядеть так:

```
tableView.beginUpdates()
tableView.insertRows(at: [IndexPath(row: 0, section: 0)], with: .automatic)
tableView.insertRows(at: [IndexPath(row: 1, section: 0)], with: .automatic)
tableView.endUpdates()
```

Теперь эти действия будут выполнены одновременно. Но важно следить, чтобы данные, которые отображаются в таблице, существовали для тех индексов, с которыми происходит действие, — иначе работа приложения будет остановлена.

Например, в таблице было 5 ячеек и 5 элементов в массиве, которые мы отображаем в этих ячейках. Элементы массива — строки. Перед тем как добавить 2 ячейки, нужно добавить 2 элемента в массив:

```
data.append("One")
data.append("Two")

tableView.beginUpdates()
tableView.insertRows(at: [IndexPath(row: 0, section: 0)], with: .automatic)
tableView.insertRows(at: [IndexPath(row: 1, section: 0)], with: .automatic)
tableView.endUpdates()
```

В данном случае приложение не будет остановлено, так как данные соответствуют предстоящим действиям.

Кастомизация UICollectionView

Создание кастомных ячеек коллекции

Создание ячеек коллекции практически не отличается от добавления ячеек таблицы. Основной особенностью ячеек коллекции является то, что у них нет контейнеров. Ячейка коллекции — это обычный **view**, который можно наполнить чем угодно. Так сделано для того, чтобы можно было создать абсолютно любую ячейку, потому что коллекции могут выглядеть совершенно по-разному.

Для создания ячейки коллекции нужно добавить класс, который будет наследоваться от **UICollectionViewCell**. В нем необходимо описать компоненты, которые будут в этой ячейке, и реализовать метод **prepareForReuse**. Он реализуется так же, как для **UITableViewCell**.

Создание supplementary view

По аналогии с таблицей у коллекции есть **header** и **footer**, но называются они **supplementary views**.

Чтобы создать такой **view**, нужно сделать наследника класса **UICollectionReusableView**. После этого описать, какие компоненты будут в этом **view**. Далее создать **xib** или прототип для этого **view**. Если **view** создан с помощью **xib**, его нужно зарегистрировать в коллекции с помощью метода **register(_ nib: UINib, forSupplementaryViewOfKind kind: String, forReuseIdentifier identifier: String)**. В качестве параметра **kind** указывается одна из констант — **UICollectionView.elementKindSectionFooter** или **UICollectionView.elementKindSectionHeader**.

Чтобы добавить **supplementary view** в коллекцию, нужно реализовать следующий метод из протокола **UICollectionViewDataSource**:

```
func collectionView(_ collectionView: UICollectionView,
                    viewForSupplementaryElementOfKind kind: String,
                    at indexPath: IndexPath) -> UICollectionReusableView {

}
```

В нем, в зависимости от параметра **kind**, нужно вернуть нужную **supplementary view**. Его можно получить с помощью метода **dequeueReusableSupplementaryView(ofKind: String, withReuseIdentifier: String, for indexPath: IndexPath)**, так как для **supplementary view** тоже действует механизм переиспользования.

В большинстве случаев реализация данного метода выглядит так:

```
if kind == UICollectionView.elementKindSectionHeader {
    let view = collectionView
        .dequeueReusableSupplementaryView(ofKind:
UICollectionView.elementKindSectionHeader,
                                          withReuseIdentifier: "id1",
                                          for: indexPath)
    // Конфигурация
    return view
} else {
    let view = collectionView
        .dequeueReusableSupplementaryView(ofKind:
UICollectionView.elementKindSectionFooter,
                                          withReuseIdentifier: "id2",
                                          for: indexPath)
    // Конфигурация
    return view
}
```

Как и в случае с ячейкой, для **header** и **footer** нужно указать размеры. Для этого в протоколе **UICollectionViewDelegateFlowLayout** есть 2 метода:

```
func collectionView(_ collectionView: UICollectionView,
                    layout collectionViewLayout: UICollectionViewLayout,
                    referenceSizeForHeaderInSection section: Int) -> CGSize

func collectionView(_ collectionView: UICollectionView,
                    layout collectionViewLayout: UICollectionViewLayout,
                    referenceSizeForFooterInSection section: Int) -> CGSize
```

В первом методе надо вернуть размер **header view**, а во втором — **footer view**. Если вернуть **CGSize.zero**, метод, запрашивающий **view**, не будет вызван.

Динамическое добавление, удаление и обновление ячеек

Работая с коллекцией, можно добавлять, удалять и обновлять ячейки и секции. Для этого используются идентичные методы, за исключением тех, что вызывают обновление. Для этого у коллекции есть такой:

```
collectionView.performBatchUpdates(_ updates: (() -> Void)?,
                                   completion: ((Bool) -> Void)? = nil)
```

В блоке **updates** выполняются изменения — добавление, удаление или обновление. Блок **completion** выполняется, когда все изменения завершились. Обновление коллекции происходит асинхронно,

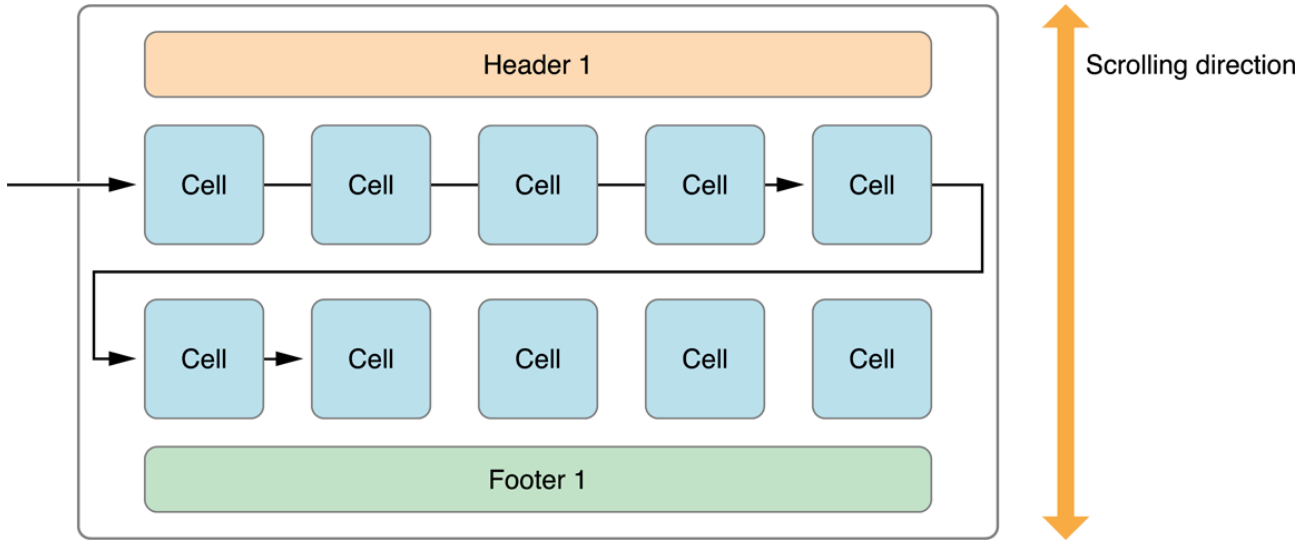
поэтому нужно следить за тем, чтобы несколько обновлений не выполнялись одновременно — иначе приложение остановится из-за ошибки.

Работа с UICollectionViewFlowLayout

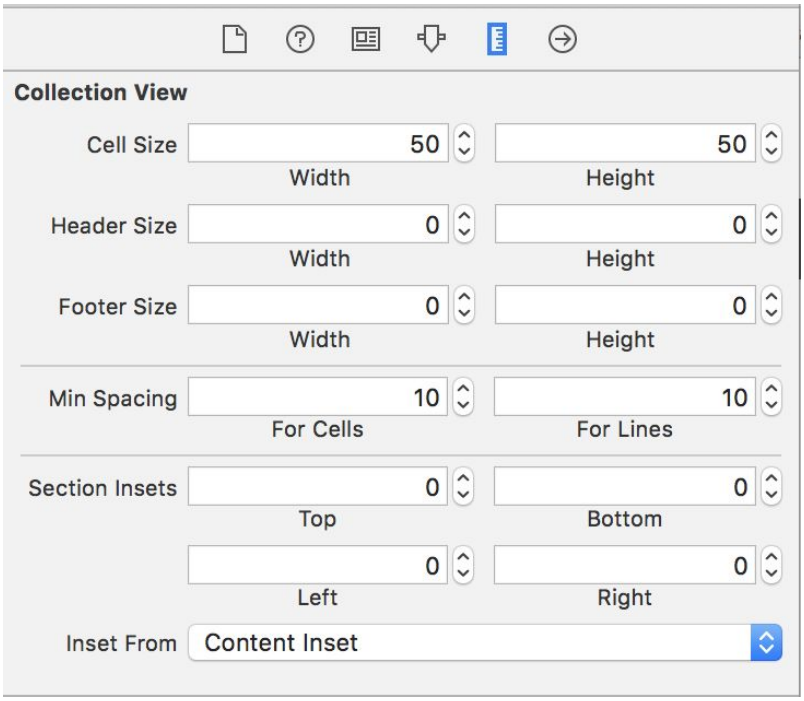
В отличие от таблицы, у коллекции есть объект, отвечающий за расположение всех элементов в ней — **layout**. Базовый класс **layout** — **UICollectionViewLayout**. Он содержит общий механизм для расстановки компонентов в коллекции. По умолчанию коллекции используют не его, а его наследника — **UICollectionViewFlowLayout**. Это **layout**, который позволяет расположить ячейки в виде таблицы с несколькими колонками.

Он бывает горизонтальным и вертикальным. Вертикальный **layout** расставляет ячейки сверху вниз, а также устанавливает вертикальное направление скролла. А горизонтальный — слева направо.

Так выглядит **flow layout**:



Можно настроить **layout,** перейдя в **interface builder** во вкладку **size inspector** и выбрав коллекцию:

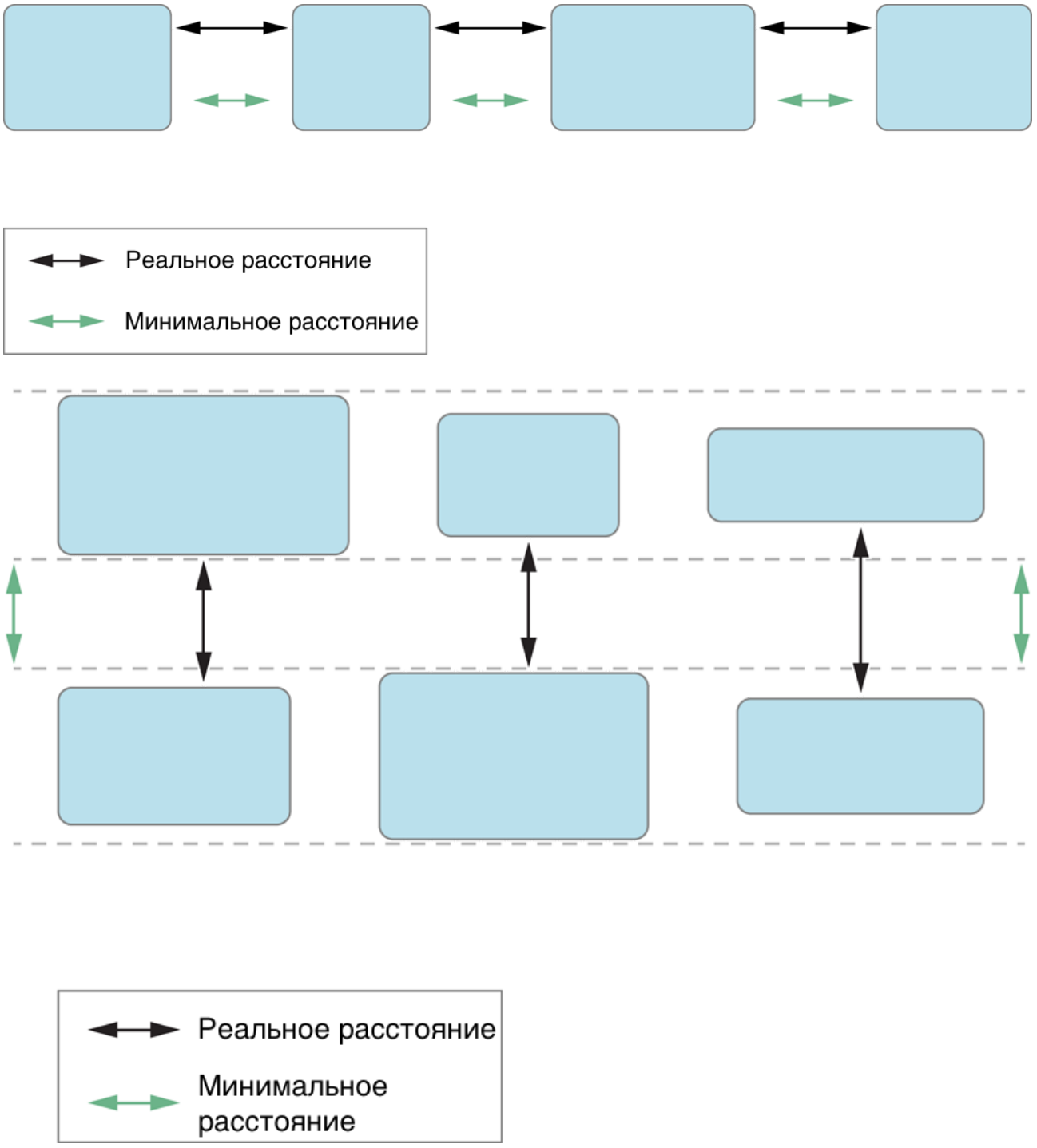


У этого **layout** есть делегат, с помощью которого его можно настраивать, — **UICollectionViewDelegateFlowLayout**. Его используют, например, когда размер ячеек зависит от экрана.

Рассмотрим методы этого делегата и что с их помощью можно настроить:

```
// Размер ячейки
func collectionView(_ collectionView: UICollectionView,
                    layout collectionViewLayout: UICollectionViewLayout,
                    sizeForItemAt indexPath: IndexPath) -> CGSize

// Отступы внутри секции
func collectionView(_ collectionView: UICollectionView,
                    layout collectionViewLayout: UICollectionViewLayout,
                    insetForSectionAt section: Int) -> UIEdgeInsets

// Минимальное расстояние между ячейками по вертикали
func collectionView(_ collectionView: UICollectionView,
                    layout collectionViewLayout: UICollectionViewLayout,
                    minimumLineSpacingForSectionAt section: Int) -> CGFloat

// Минимальное расстояние между ячейками по горизонтали
func collectionView(_ collectionView: UICollectionView,
                    layout collectionViewLayout: UICollectionViewLayout,
                    minimumInteritemSpacingForSectionAt section: Int) -> CGFloat

// Размер header view
func collectionView(_ collectionView: UICollectionView,
                    layout collectionViewLayout: UICollectionViewLayout,
                    referenceSizeForHeaderInSection section: Int) -> CGSize

// Размер footer view
func collectionView(_ collectionView: UICollectionView,
                    layout collectionViewLayout: UICollectionViewLayout,
                    referenceSizeForFooterInSection section: Int) -> CGSize
```

Остановимся подробнее на методах, которые возвращают минимальные расстояния между ячейками по горизонтали и вертикали. Это расстояние служит граничным значением для **flow layout**. Если у ячеек разный размер, расстояние будет не меньше минимального. **Flow layout** растягивает все ячейки, которые помещаются в линию, поэтому реальное расстояние между ними будет больше минимального. По вертикали может быть так же, ведь высота линии зависит от высоты самой большой ячейки, а все остальные ставятся в центр линии по вертикали:



Создание кастомного layout коллекции

Не всегда **UICollectionViewFlowLayout** соответствует требованиям к дизайну, поэтому приходится создавать свой **layout**. Для этого надо добавить наследника класса **UICollectionViewLayout** и переопределить следующие его методы:

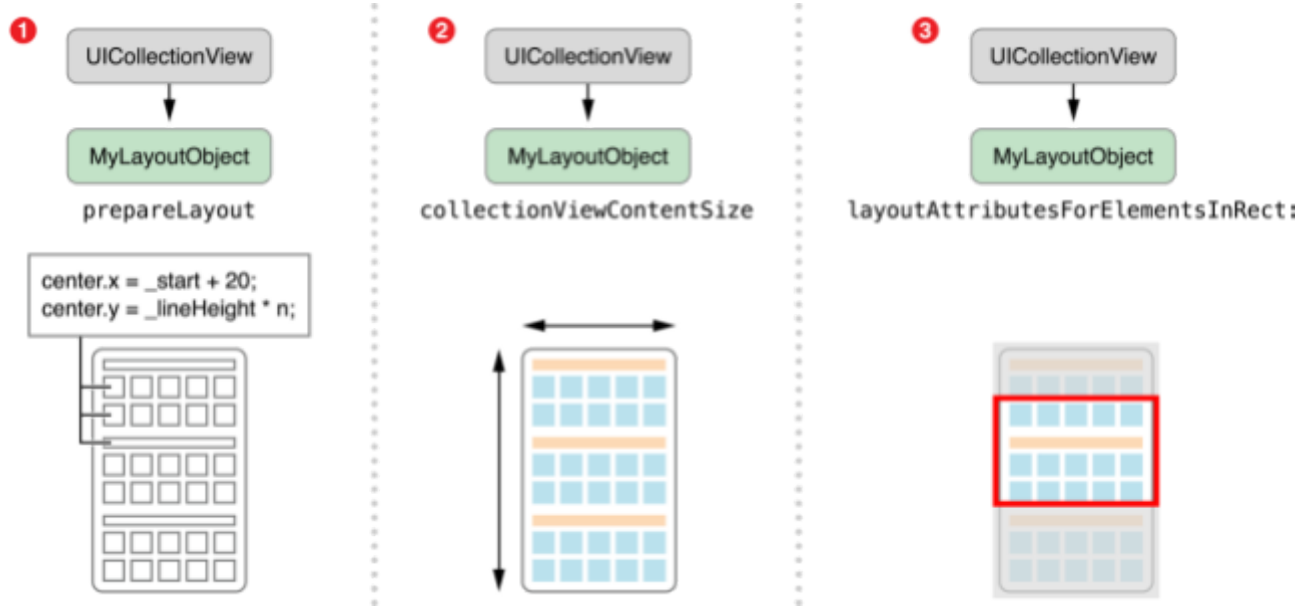
- **prepare()** — метод, в который используется для расчета атрибутов для всех ячеек;
- **layoutAttributesForElements(in rect: CGRect) -> [UICollectionViewLayoutAttributes]?** — массив атрибутов ячеек, находящихся в заданном прямоугольнике;

- **collectionViewContentSize: CGSize** — вычисляемая переменная, которая возвращает размер контента коллекции.

Эти методы вызываются коллекцией и дают понять объекту **layout**, как необходимо отображать ячейки.

UICollectionViewLayoutAttributes — набор атрибутов, таких как **frame**, **center**, **alpha**, **zPosition**. Они применяются к ячейкам, чтобы организовать **layout**. Это информация, как должны быть расположены ячейки в коллекции.

Схематически процесс работы **layout** выглядит так:



Сначала рассчитываем атрибуты для всех ячеек коллекции в методе **prepare**, затем возвращаем размер контента и необходимые атрибуты для запрошенной прямоугольной области.

Практика

Создание ячейки для списка городов

У нас уже есть таблица со списком городов, но она состоит из стандартных ячеек. Создадим кастомную ячейку, которая будет содержать название города, его герб и градиент на фоне.

Сначала создадим новый файл **swift** и добавим туда класс нашей ячейки, **IBOutlet's** для лейбла с названием города и **UIImageView** для герба. Также в методе **didSet** каждого свойства напишем настройку дизайна для компонента:

```
class AllCitiesCell: UITableViewCell {

    @IBOutlet var cityTitleLabel: UILabel! {
        didSet {
            self.cityTitleLabel.textColor = UIColor.yellow
        }
    }
    @IBOutlet var cityEmblemView: UIImageView! {
        didSet {
            self.cityEmblemView.layer.borderColor = UIColor.white.cgColor
            self.cityEmblemView.layer.borderWidth = 2
        }
    }

}
```

Лейблу с названием города мы установили белый цвет, а **UIImageView** с гербом добавили обводку белого цвета.

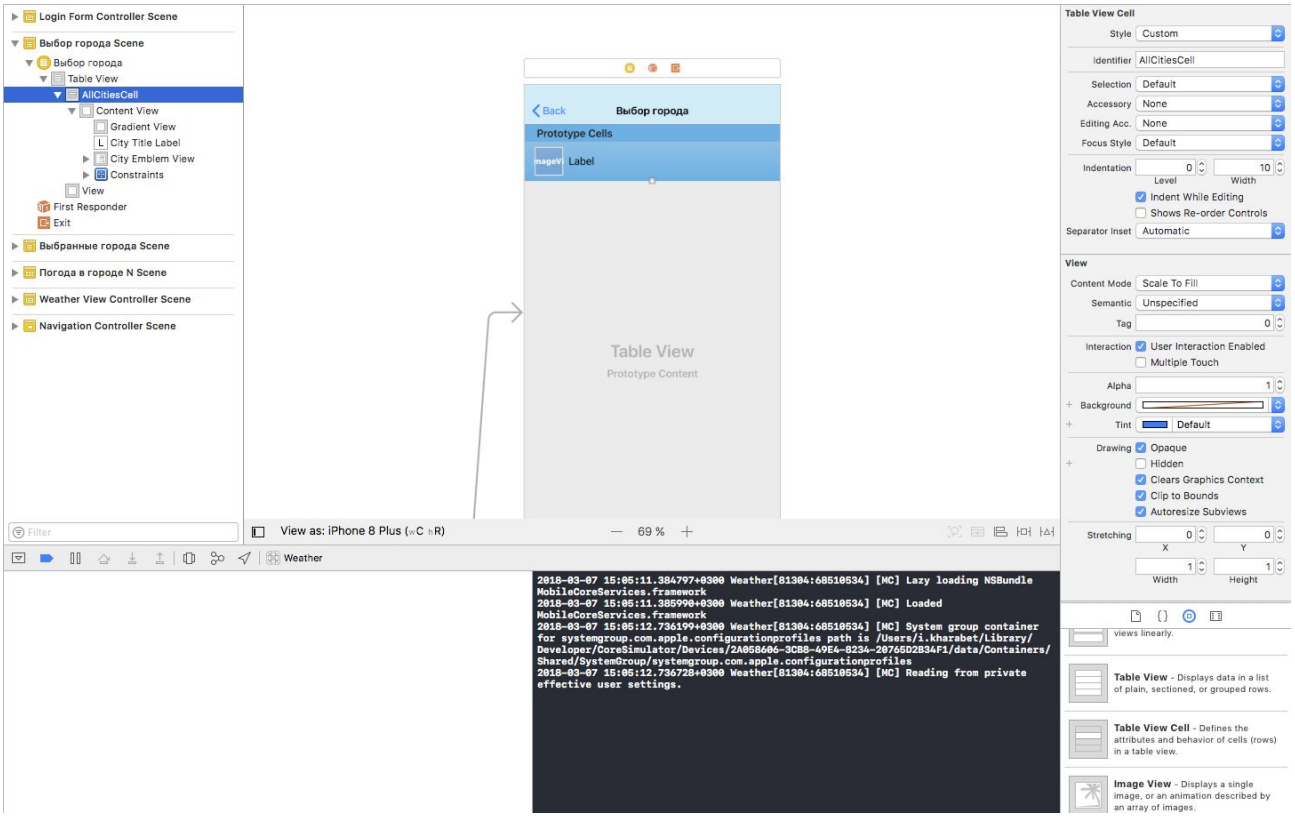
Теперь необходимо написать метод для конфигурации ячейки с переданными параметрами и переопределить метод **prepareForReuse**:

```
func configure(city: String, emblem: UIImage) {
    self.cityTitleLabel.text = city
    self.cityEmblemView.image = emblem

    self.backgroundColor = UIColor.black
}

override func prepareForReuse() {
    super.prepareForReuse()
    self.cityTitleLabel.text = nil
    self.cityEmblemView.image = nil
}
```

Осталось сделать закругление картинки с гербом:

```
override func layoutIfNeeded() {
    super.layoutIfNeeded()

    cityEmblemView.clipsToBounds = true
    cityEmblemView.layer.cornerRadius = cityEmblemView.frame.width / 2
}
```

Теперь можно создать прототип ячейки в **storyboard**, добавить необходимые UI-компоненты и настроить их дизайн:



Теперь нам надо как-то хранить эмблемы для городов. Мы используем кортеж.

```
var cities = [
        (title: "Moscow", emblem:  imageLiteral(resourceName: "moskow")),
        (title: "Krasnoyarsk", emblem:  imageLiteral(resourceName:
"krasnoyarsk")),
        (title: "London", emblem:  imageLiteral(resourceName: "london")),
        (title: "Paris", emblem:  imageLiteral(resourceName: "paris")),
        (title: "Nobos", emblem:  imageLiteral(resourceName: "paris"))
    ]
```

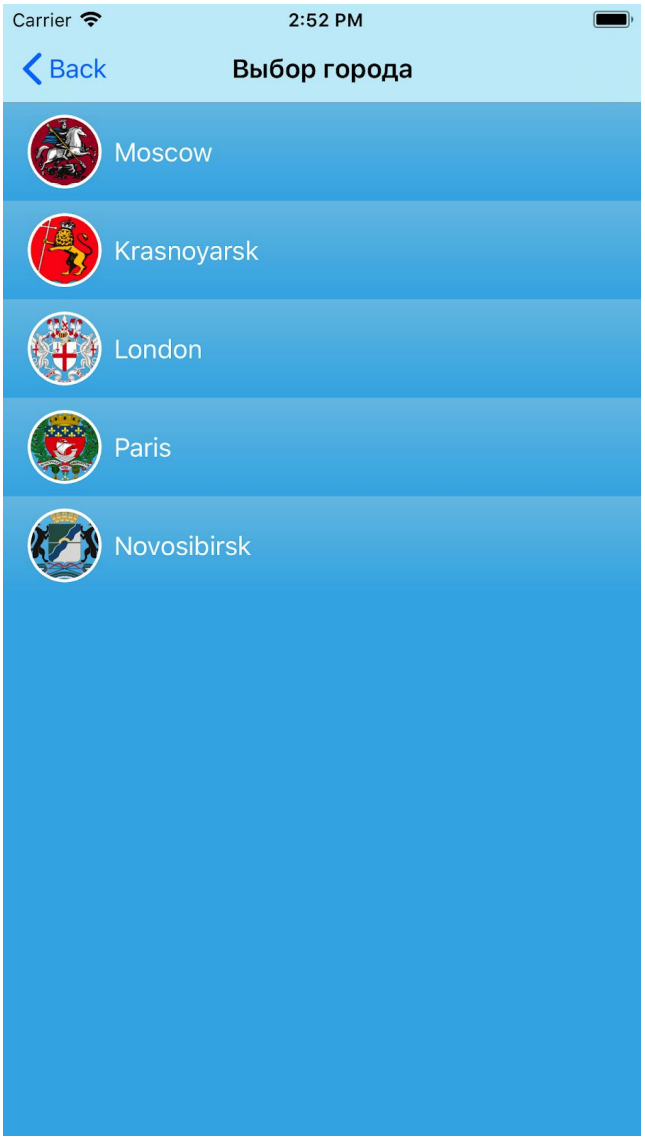
Следует так же изменить реализацию метода **tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell**:

```
let cell = tableView.dequeueReusableCell(withIdentifier: "AllCitiesCell", for:
indexPath) as! AllCitiesCell

let city = self.cities[indexPath.row]
cell.configure(city: city.title, emblem: city.emblem)

return cell
```

И реализацию метода **addCity(segue: UIStoryboardSegue)** в классе **MyCitiesController.**

```
@IBAction func addCity(segue: UIStoryboardSegue) {

        // Проверяем идентификатор перехода, чтобы убедиться, что это нужный
        if segue.identifier == "addCity" {
            // Получаем ссылку на контроллер, с которого осуществлен переход
            let allCitiesController = segue.source as! AllCitiesController

            // Получаем индекс выделенной ячейки
            if let indexPath =
allCitiesController.tableView.indexPathForSelectedRow {
                // Получаем город по индексу
                let city = allCitiesController.cities[indexPath.row]

                // Проверяем, что такого города нет в списке
                if !cities.contains(city.title) {
                    // Добавляем город в список выбранных
                    cities.append(city.title)
                    // Обновляем таблицу
                    tableView.reloadData()
                }
            }
        }
    }
```

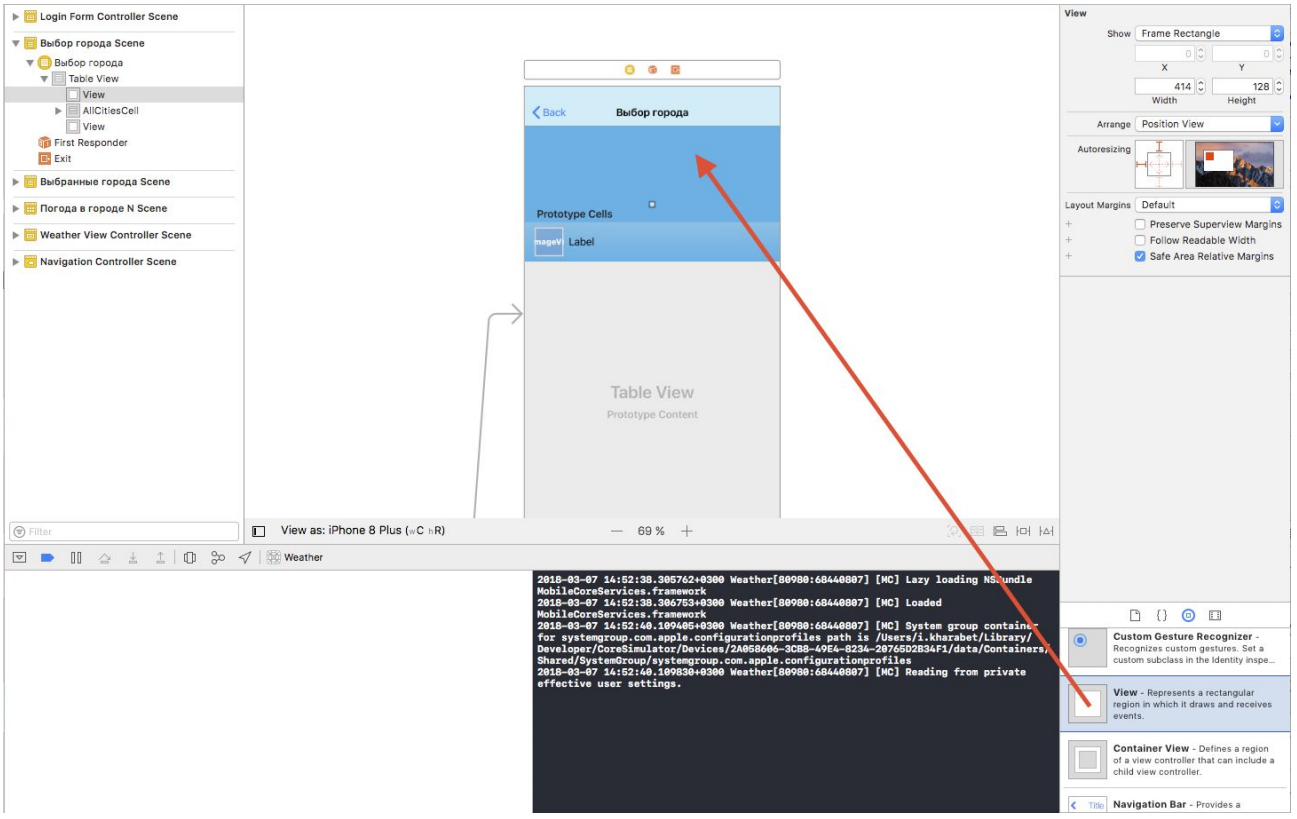
После этого можно запустить проект и посмотреть результат:



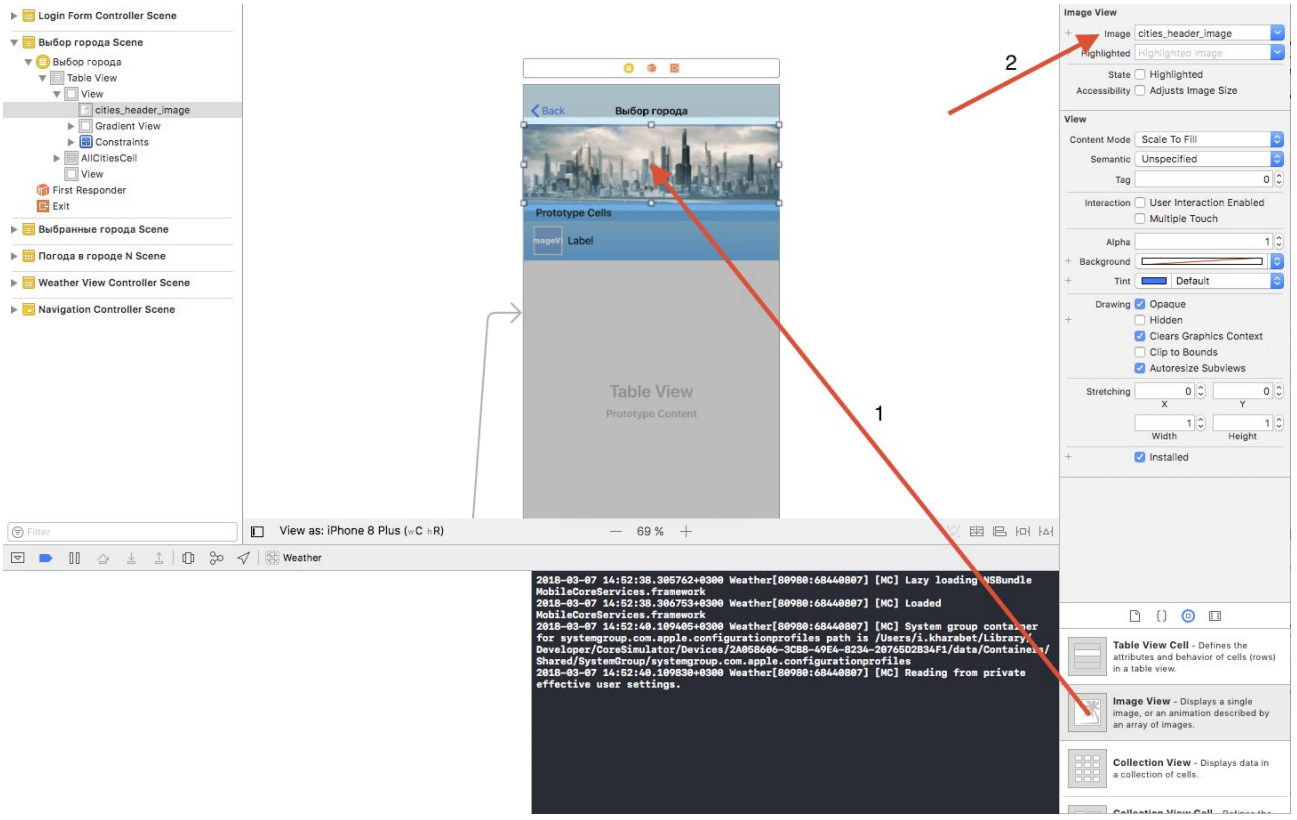
Создание header view для списка городов

На том же экране выбора города добавим **header view** для таблицы. Он будет иметь декоративную функцию: содержать изображение города и небольшой градиент.

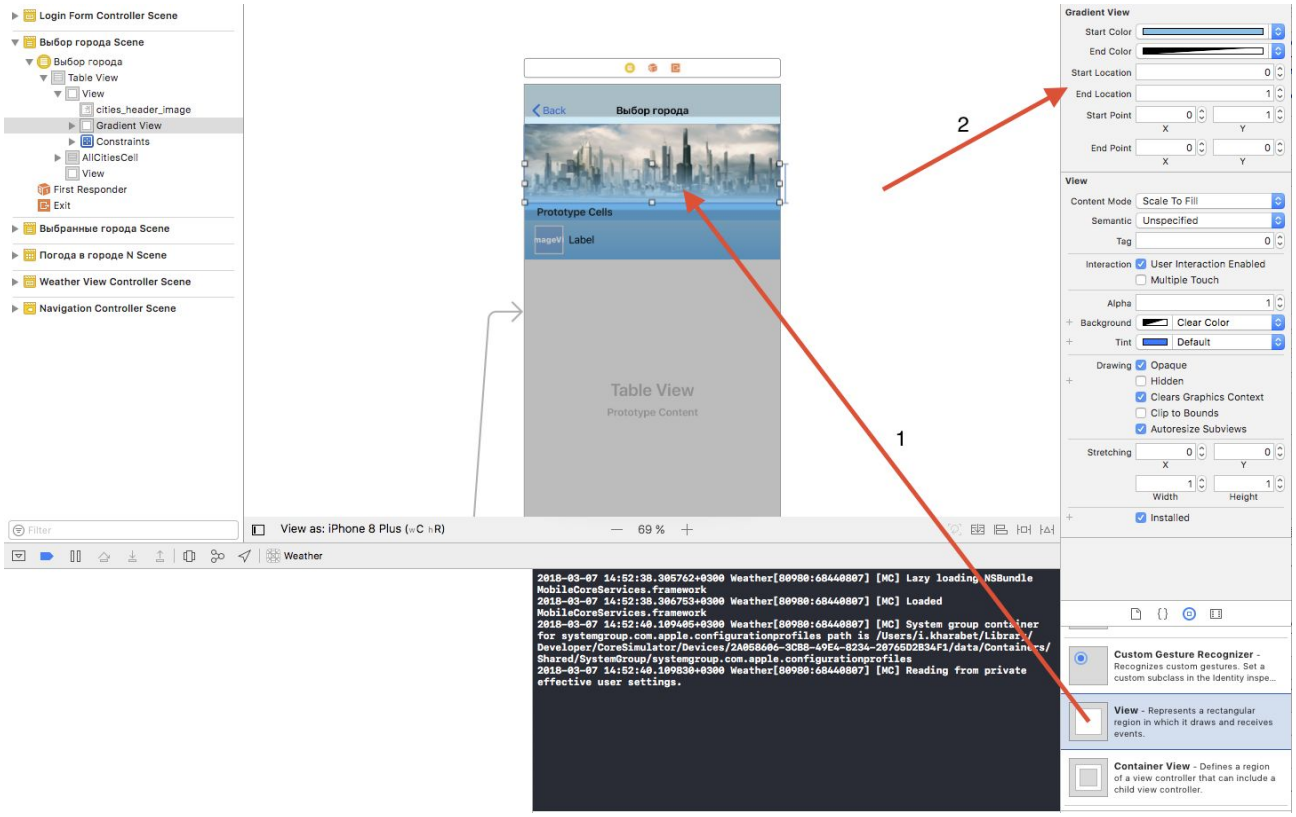
Добавим пустой **UIView** в **header** таблицы и зададим ему размер 128 точек. Сделать это можно, перетащив объект **UIView** чуть выше первого прототипа ячейки:



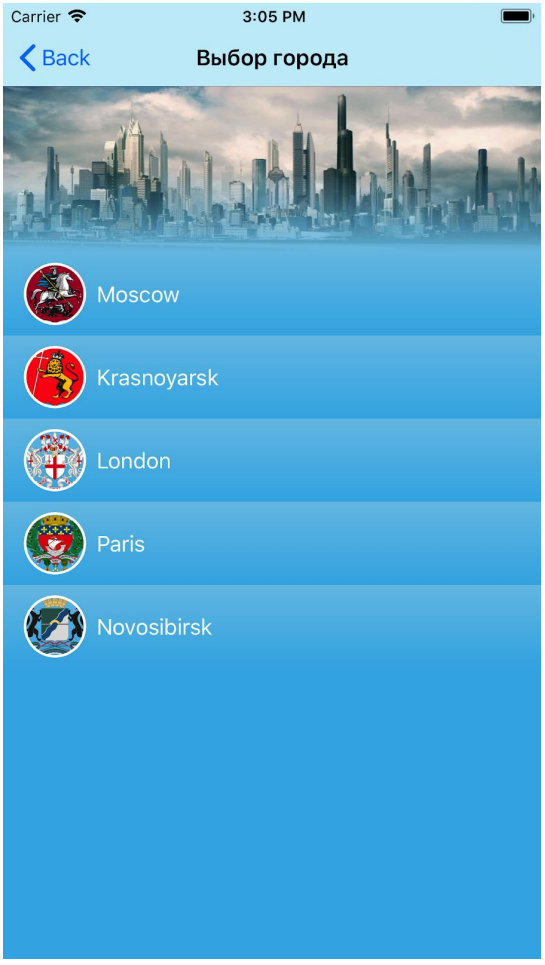
Далее нужно добавить в этот **view UIImageView** и установить ему картинку:



После этого добавим **GradientView** и снизу — **header view**. Зададим **GradientView** размер, равный половине **header view**:


Login Form Controller Scene
Выбор города Scene
Выбор города
Table View
cities_header_image
Gradient View
Constraints
AllCitiesCell
View
First Responder
Exit
Выбранные города Scene
Погода в городе N Scene
Weather View Controller Scene
Navigation Controller Scene
Back
Выбор города
Prototype Cells
Label
Table View
Prototype Content
1
2
Gradient View
Start Color
End Color
Start Location
End Location
Start Point
End Point
View
Content Mode
Scale To Fill
Semantic
Unspecified
Tag
Interaction
User Interaction Enabled
Multiple Touch
Alpha
Background
Clear Color
Tint
Default
Drawing
Opaque
Hidden
Clears Graphics Context
Clip to Bounds
Autoresize Subviews
Stretching
Width
Height
Installed
Custom Gesture Recognizer - Recognizes custom gestures. Set a custom subclass in the Identity inspe...
View - Represents a rectangular region in which it draws and receives events.
Container View - Defines a region of a view controller that can include a child view controller.
View as: iPhone 8 Plus
69 %
Weather
2018-03-07 14:52:38.305762+0300 Weather[80980:68440807] [MC] Lazy loading NSBundle MobileCoreServices.framework
2018-03-07 14:52:38.306753+0300 Weather[80980:68440807] [MC] Loaded MobileCoreServices.framework
2018-03-07 14:52:40.109405+0300 Weather[80980:68440807] [MC] System group container for systemgroup.com.apple.configurationprofiles path is /Users/i.kharabet/Library/Developer/CoreSimulator/Devices/2A058606-3CB8-49E4-8234-20765D2B34F1/data/Containers/Shared/SystemGroup/systemgroup.com.apple.configurationprofiles
2018-03-07 14:52:40.109830+0300 Weather[80980:68440807] [MC] Reading from private effective user settings.

Результат:

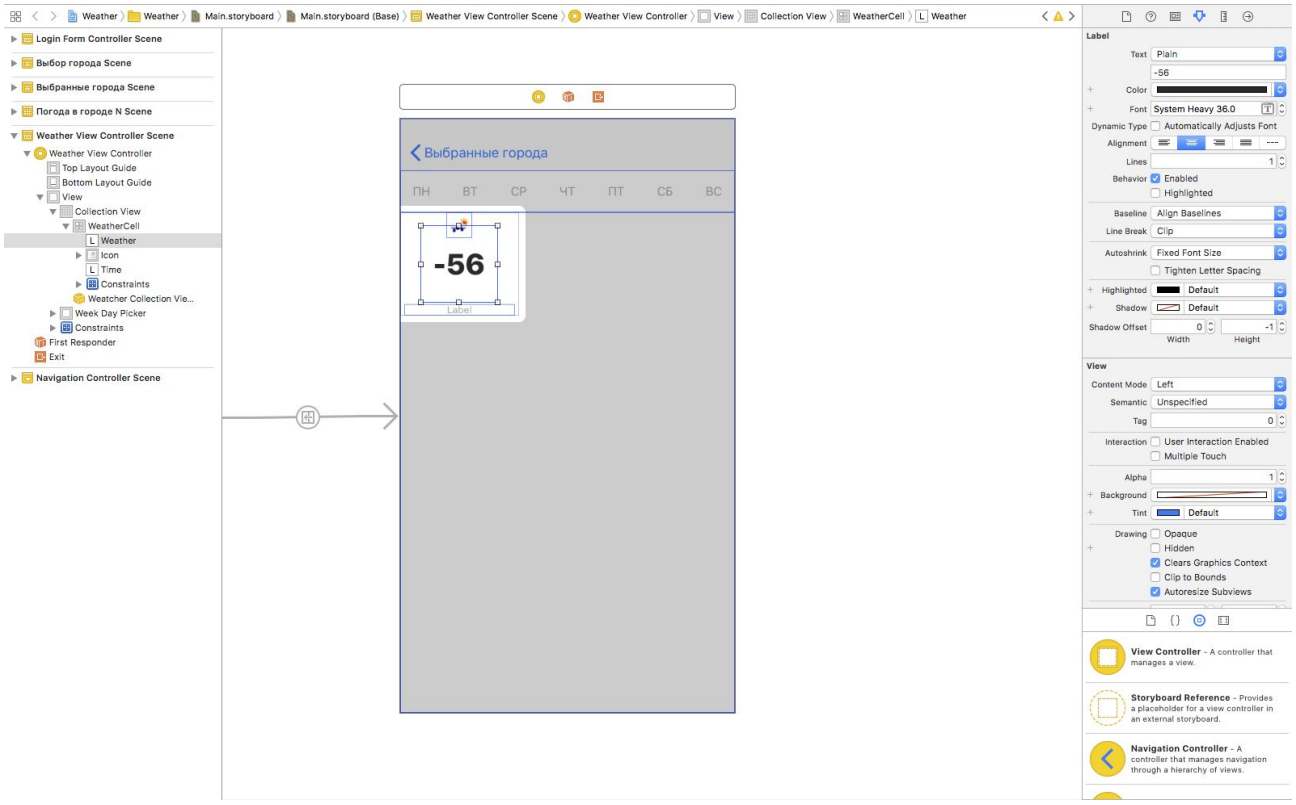

Carrier
3:05 PM
Back
Выбор города
Moscow
Krasnoyarsk
London
Paris
Novosibirsk

Создание ячейки для коллекции на экране погоды

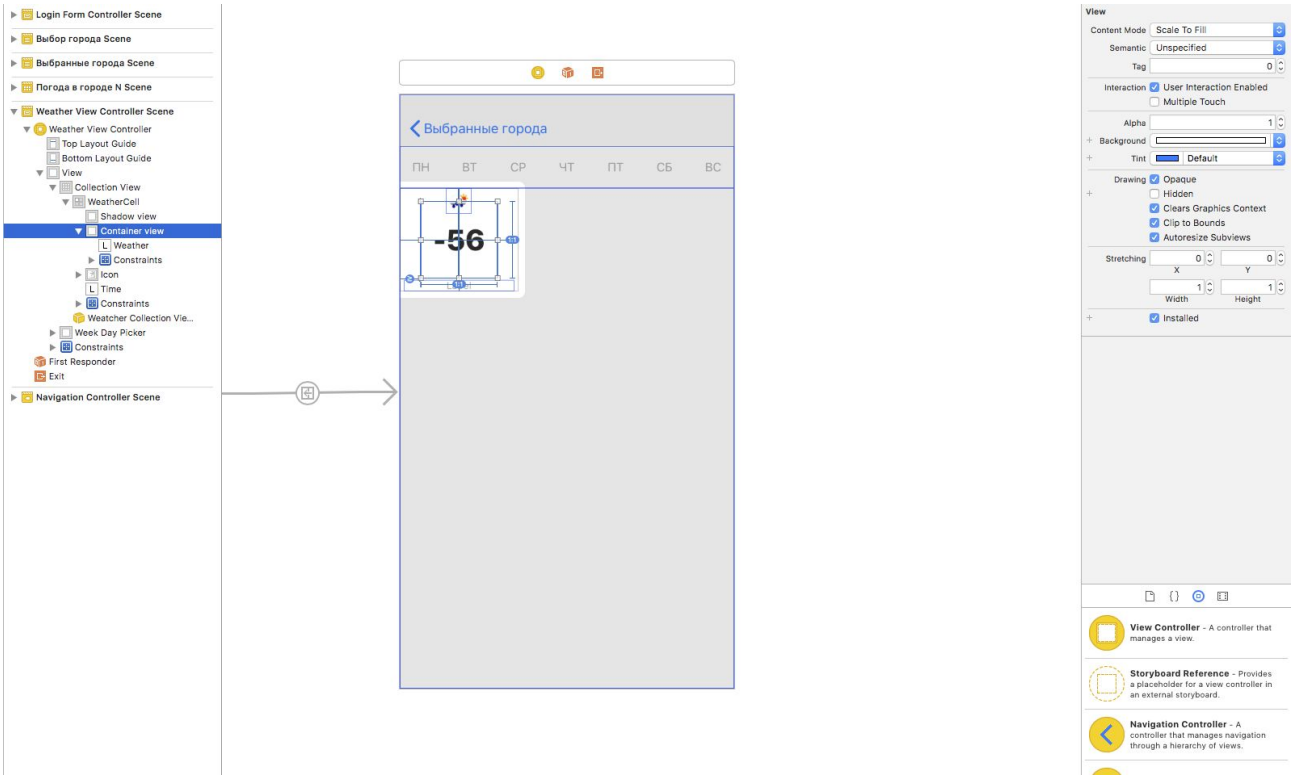
У нас уже есть ячейка для коллекции погоды — **WeatherCell**. Улучшим ее дизайн:

В новой ячейке поменялось расположение элементов, выделена температура и добавлена тень, цвет которой зависит от погоды.

Так как все три элемента в ячейке остались, достаточно просто поменять их расположение и параметры:



Появилось два новых элемента — круглый **view** и тень под ним. Добавим их в ячейку:



Теперь перейдем к коду. Добавим два **IBOutlet** для новых **view** и настроим их в методе **didSet**:

```
@IBOutlet weak var shadowView: UIView! {
    didSet {
        self.shadowView.layer.shadowOffset = .zero
        self.shadowView.layer.shadowOpacity = 0.75
        self.shadowView.layer.shadowRadius = 6
        self.shadowView.backgroundColor = .clear
    }
}

@IBOutlet weak var containerView: UIView! {
    didSet {
        self.containerView.clipsToBounds = true
    }
}
```

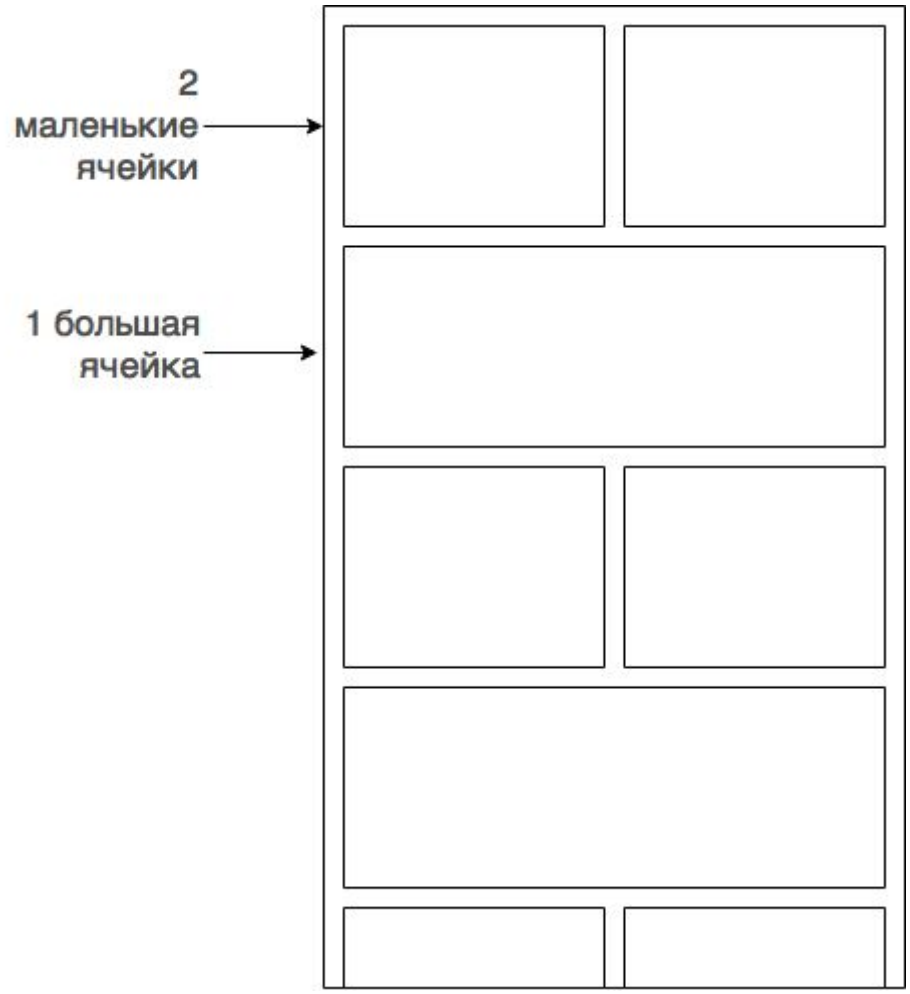
Далее нужно сделать круглые **view** и тень. Для этого в методе **layoutSubviews** напишем следующий код:

```
override func layoutSubviews() {
    super.layoutSubviews()

    self.shadowView.layer.shadowPath = UIBezierPath(ovalIn:
self.shadowView.bounds).cgPath
    self.containerView.layer.cornerRadius = self.containerView.frame.width / 2
}
```

На этом разработка ячейки окончена.

Создание кастомного layout для коллекции на экране погоды

Кастомный layout для коллекции на экране погоды будет выглядеть так:



Создадим новый класс — **WeatherCollectionViewLayout**, который будет наследником **UICollectionViewLayout**:

```
class WeatherCollectionViewLayout: UICollectionViewLayout {

}
```

Добавим в этот класс свойства, необходимые для дальнейшей работы:

```
var cacheAttributes = [IndexPath: UICollectionViewLayoutAttributes]()
                                        // Хранит атрибуты для заданных индексов

var columnsCount = 2                    // Количество столбцов

var cellHeight: CGFloat = 128           // Высота ячейки

private var totalCellsHeight: CGFloat = 0 // Хранит суммарную высоту всех ячеек
```

После этого нужно переопределить метод **prepare**. Это самая сложная часть в разработке кастомного **layout**, поэтому разберем ее поэтапно. Сначала переопределим сам метод и добавим несколько проверок:

```
override func prepare() {
    self.cacheAttributes = [:] // Инициализируем атрибуты

    // Проверяем наличие collectionView
    guard let collectionView = self.collectionView else { return }

    let itemsCount = collectionView.numberOfItems(inSection: 0)
    // Проверяем, что в секции есть хотя бы одна ячейка
    guard itemsCount > 0 else { return }

}
```

Далее нужно определить ширину для большой ячейки и для маленькой:

```
let bigCellWidth = collectionView.frame.width
let smallCellWidth = collectionView.frame.width / CGFloat(self.columnsCount)
```

Затем нужно посчитать **frame** для каждой ячейки. Для этого добавим цикл, в котором будем создавать атрибуты для ячейки и считать **frame**, а также переменные **lastX** и **lastY**, которые будут хранить последние **x** и **y** для вычисления **origin** каждого **frame**:

```
var lastY: CGFloat = 0
var lastX: CGFloat = 0

for index in 0..<itemsCount {
    let indexPath = IndexPath(item: index, section: 0)
    let attributes = UICollectionViewLayoutAttributes(forCellWith: indexPath)

}
```

Теперь вычисление **frame** разделится на две части — для большой и для маленькой ячейки. Чтобы определить, большая ячейка или маленькая, достаточно взять остаток от деления (текущего индекса + 1) на (количество столбцов + 1). Если остаток равен 0, это большая ячейка. Добавим вычисление в код:

```
let isBigCell = (index + 1) % (self.columnsCount + 1) == 0
```

Напишем вычисление **frame** в зависимости от ячейки:

```
if isBigCell {
    attributes.frame = CGRect(x: 0, y: lastY,
                              width: bigCellWidth, height: self.cellHeight)

} else {
    attributes.frame = CGRect(x: lastX, y: lastY,
                              width: smallCellWidth, height: self.cellHeight)

}
```

Теперь нужно присвоить значения переменным **lastX** и **lastY**, для этого модифицируем предыдущий код:

```
if isBigCell {
    attributes.frame = CGRect(x: 0, y: lastY,
                              width: bigCellWidth, height: self.cellHeight)

    lastY += self.cellHeight
} else {
    attributes.frame = CGRect(x: lastX, y: lastY,
                              width: smallCellWidth, height: self.cellHeight)

    let isLastColumn = (index + 2) % (self.columnsCount + 1) == 0 || index ==
itemsCount - 1
    if isLastColumn {
        lastX = 0
        lastY += self.cellHeight
    } else {
        lastX += smallCellWidth
    }
}
```

После этого добавим атрибуты в словарь:

```
cacheAttributes[indexPath] = attributes
```

И присвоим значение переменной **totalCellsHeight**. Оно будет равно **lastY**:

```
self.totalCellsHeight = lastY
```

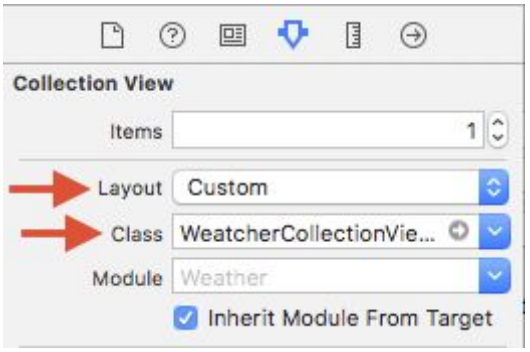
Метод **prepare** готов. Теперь нужно переопределить методы, которые возвращают атрибуты для заданных области и индекса:

```
override func layoutAttributesForElements(in rect: CGRect) ->
[UICollectionViewLayoutAttributes]? {
    return cacheAttributes.values.filter { attributes in
        return rect.intersects(attributes.frame)
    }
}

override func layoutAttributesForItem(at indexPath: IndexPath) ->
UICollectionViewLayoutAttributes? {
    return cacheAttributes[indexPath]
}
```

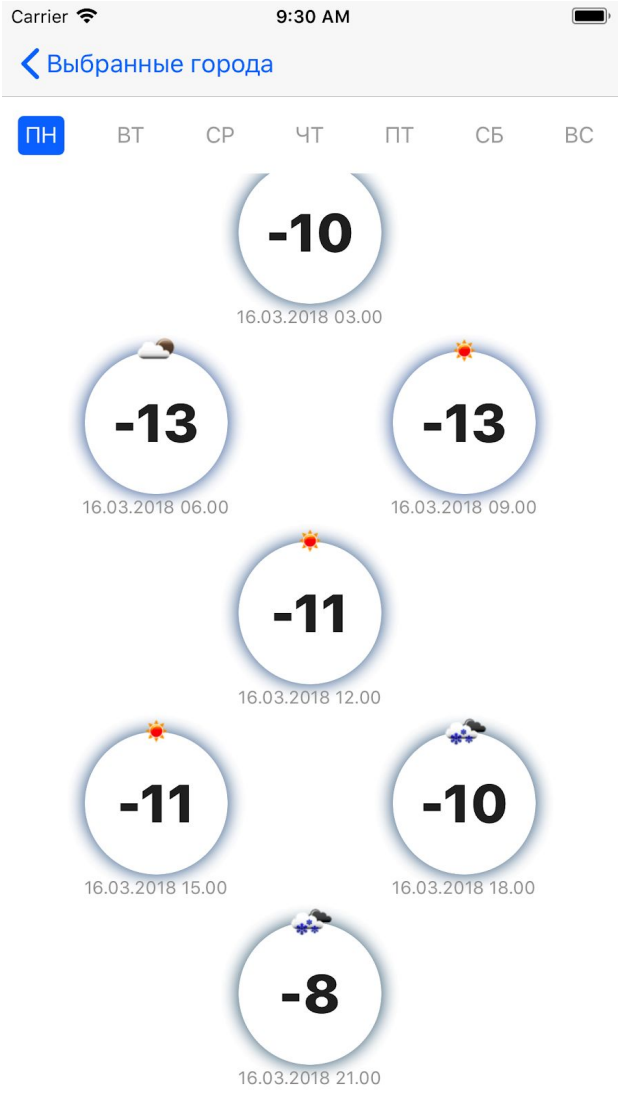
Осталось переопределить свойство **collecitonViewContentSize**:

```
override var collectionViewContentSize: CGSize {
    return CGSize(width: self.collectionView?.frame.width ?? 0,
                  height: self.totalCellsHeight)
}
```

На этом разработка **layout** закончена. Установим его для коллекции:



В результате коллекция будет выглядеть так:



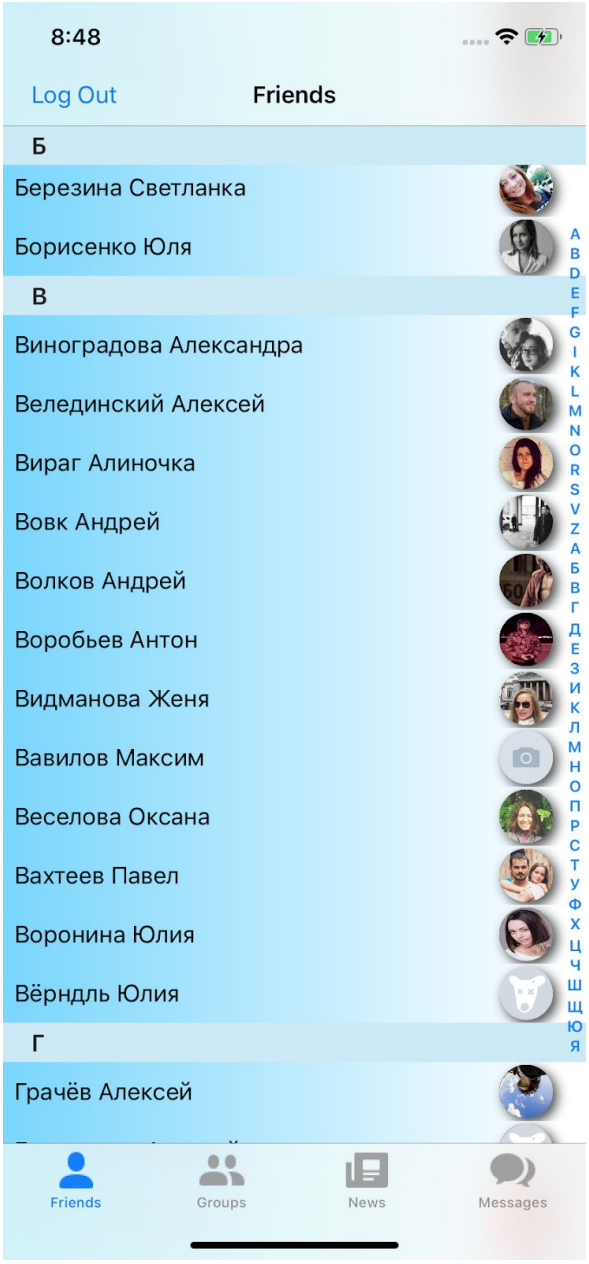
Практическое задание

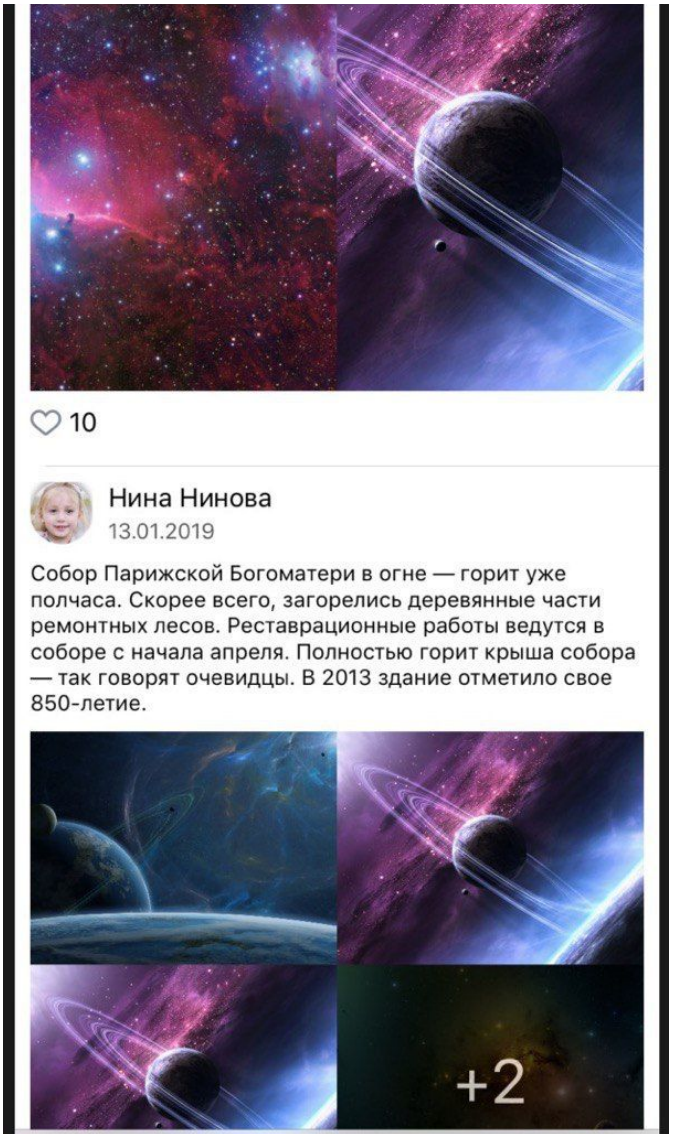
На основе предыдущего ПЗ.

1. Сделать группировку друзей по первой букве фамилии. Добавить **header** секции для таблицы со списком друзей. Он должен содержать первую букву фамилии и иметь полупрозрачный фон, совпадающий по цвету с таблицей.
2. Добавить **UISearchBar** в **header** таблицы со списком друзей и групп. Указать контроллер, содержащий эту таблицу, делегатом **UISearchBar**, реализуйте поиск с выводом результатов в ту же таблицу. Для простоты реализации не стоит использовать **UISearchController**. (Задание на самостоятельный поиск решения.)
3. Создать экран новостей. Добавить туда таблицу и сделать ячейку для новости. Ячейка должна содержать то же самое, что и в оригинальном приложении «ВКонтакте»: надпись, фотографии, кнопки «Мне нравится», «Комментировать», «Поделиться» и индикатор количества просмотров. Сделать поддержку только одной фотографии, которая должна быть квадратной и растягиваться на всю ширину ячейки. Высота ячейки должна вычисляться автоматически.
4. * В ячейку новости добавить отображение нескольких фотографий. Они должны располагаться в квадратной зоне, по ширине равной ячейке. В идеале нужно равномерно

расположить фотографии в квадратной области. (Необязательное задание — для тех, у кого есть время.)

Примеры выполненных работ





Дополнительные материалы

1. Anatomy of a TableView(Controller) Architecture.
2. A Closer Look at Table View Cells.
3. UICollectionView Custom Layout Tutorial: Pinterest.
4. Custom Layouts: A Worked Example.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. A Closer Look at Table View Cells.
2. Collection View Basics.
3. Designing Your Data Source and Delegate.
4. Using the Flow Layout.
5. Creating Custom Layouts.