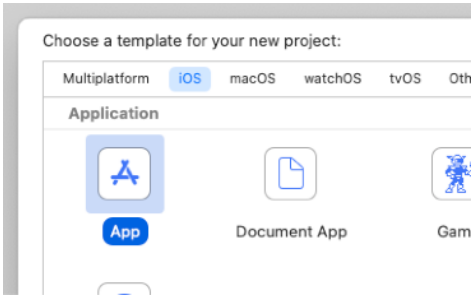
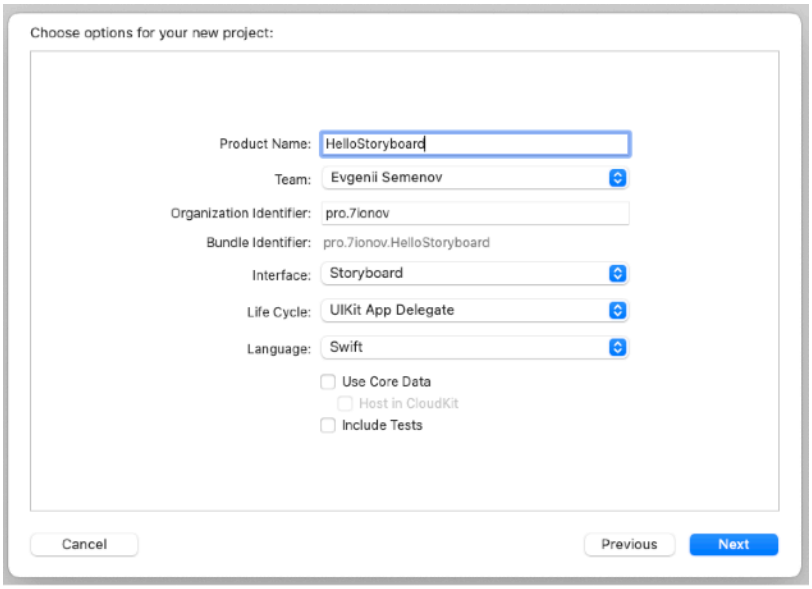
Интерфейс Xcode. Работа со .storyboard и .xib файлами

Начнем с создания простого приложения под систему iOS. Для этого выбираем создание проекта любым удобным способом (через стартовое окно Xcode, либо через меню File>New>Project). В качестве платформы выбираем iOS, а в качестве шаблона проекта выбираем стандартное приложение (App). Выбрав данный шаблон мы получим уже настроенный проект с пока что пустым экраном, но который уже готов к запуску и нам не нужно будет дополнительно что-то настраивать.



Следующим шагом нужно определиться с названием проекта. То, какое название вы дадите своему проекту повлияет только на то, как будет отображаться название папки вашего проекта. Название самого приложения, которое будет подписано под его иконкой можно будет легко изменить позже. Здесь есть пара важных моментов, на которых я хотел бы остановиться. Во-первых, не стоит называть свой проект кириллицей или использовать какие-то специфичные символы. В будущем будет вероятность получить какую-то непредвиденную ошибку и потратить на неё ни один день, хотя вся проблема будет в одном символе, содержащемся в пути к проекту (например при добавлении сторонних библиотек в ваш проект). Во-вторых, не используйте для проекта типовые названия, которые в будущем могут пересекаться с названиями других библиотек или систем, добавляемых в ваш проект. Здесь шанс попасть в точное название не так велик, но всё же старайтесь дать проекту осмысленное индивидуальное название, так вы избежите проблем. Конечно, позже у вас будет возможность переименовать проект. В интернете есть ни одна статья на этот счет. Но учтите, что переименовать проект, эта значит обновить имя во всех местах, где оно используется, а не только в названии папки и файла вашего проекта. Так, что даже 1 пропущенный файл негативно скажется на работе приложения, поэтому лучше сразу придумать такое имя, которое вас будет полностью устраивать. Но не нужно подходить к этому слишком ответственно и ломать голову.

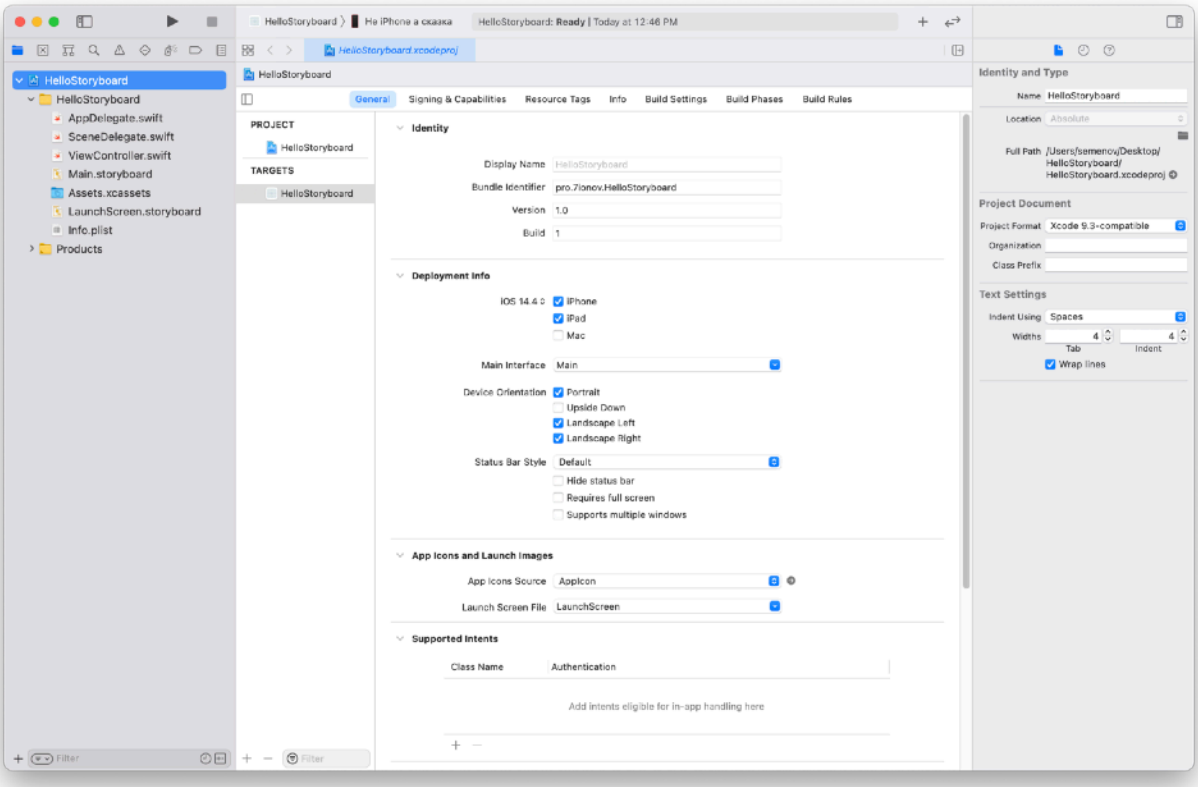


После названия нам предлагается выбрать команду (разработчика), который создает проект. Это будет играть для нас роль, когда мы захотим запустить приложение на реальном устройстве, либо отправить приложение в магазин приложений. Вы также сможете сменить это значение в будущем. Если у вас не

отображается никакой команды, нужно зарегистрироваться на сайте https://developer.apple.com в качестве разработчика (бесплатно), используя ваш appleID, а после добавить данный аккаунт в настройках Xcode. Вообще есть два варианта лицензий разработчика. Бесплатная (которая нас интересует) просто дает доступ к различным сертификатам, позволяющем запустить наше приложение на телефоне, а также к документации, тестовым версиям программ для разработки и различным вспомогательным утилитам для тестирования приложения. Также есть платная лицензия по $99 в год, которая позволяет размещать приложения в магазине и дает больше возможностей для использования сервисов (например, использование пуш-уведомлений доступно только для оплаченной лицензии разработчика).

Далее указываем уникальный идентификатор приложения, он может быть совершенно любым, главное, чтобы он был уникальным. Можете здесь указывать свое имя или адрес сайта, если он есть (обычно указывают адрес с конца наоборот, чтоб не путать с обычным url-адресом).

В качестве последних шагов нужно определиться с используемыми технологиями для создания шаблона приложения. Сейчас нас интересует интерфейс через Storyboard, потому оставляем этот пункт, а библиотека для создания UI-интерфейса приложения UIKit, о ней мы и будем говорить в дальнейшем. В качестве языка оставляем Swift, а галочки ниже не ставим. Они включают шаблонный код для тестирования приложения и организации хранения информации в базе данных, что пока нам не нужно. Нажимаем далее и выбираем место сохранения проекта.



После всех манипуляций с настройкой проекта у нас откроется окно Xcode с проектом. Рассмотрим подробнее из каких блоков оно состоит. Верхняя панель включает в себя следующие элементы:

- Кнопки запуска и остановки текущего проекта
- Выбранный проект для запуска (если состоит из нескольких модулей) и устройство или симулятор (добавлять симуляторы можно здесь же через последний пункт в списке)

- Строка текущего состояния проекта и информация о количестве предупреждений, и ошибок
- Библиотека шаблонов и ресурсов (цвета, картинки) проекта в коде, либо UI элементов для storyboard файлов
- Режим версионирования (полезно сравнивать изменения в коде при работе с git)
- Кнопки для отображения/скрытия боковых панелей (слева и справа)

Весь функционал программы Xcode может быть настроен на горячие клавиши. По умолчанию уже идет много удобных комбинаций (например cmd+R для запуска проекта, cmb+B для сборки проекта и проверки на ошибки, cmd+shift+L для открытия окна библиотеки шаблонов), а остальные можно задать в настройках программы.

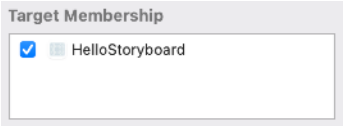
Левая панель состоит из следующих разделов:

- Навигатор по файлам и папкам проекта
- Навигатор системы управления версиями (при использовании git)
- Навигатор по классам всего проекта
- Навигатор глобального поиска по проекту
- Навигатор по ошибкам и предупреждениям в проекте
- Навигатор по тестам (когда в проекте будет тестирование функциональности)
- Навигатор отладки (во время запуска проекта здесь будет отображаться информация о текущих процессах и потребляемых ресурсах)
- Навигатор по точкам останова (так называемые breakpoints, иначе говоря возможность поставить на паузу выполнение кода в любом месте, чтобы что-то проверить, например, что в программе верно вычисляются некоторые величины)
- Навигатор отчетов (информация о всех запусках проекта)

Быстрый доступ к разделам левой панели можно осуществлять через зажатие cmd и нажатие цифры с порядковым номером раздела (например cmd+4 для поиска), либо через дополнительные комбинации, настроенные для Xcode (для того же поиска есть комбинация cmd+shift+F). Также удобными комбинациями являются: поиск файла проекта через cmd+shift+O (когда у вас очень много файлов в проекте) и отображение текущего открытого файла в разделе файловой системы cmd+shift+J (если к примеру вы нашли файл через поиск и хотите посмотреть, где он находится в проекте и какие файлы есть рядом с ним).

Разделы правой панели зависят от того, что сейчас выбрано в качестве рабочего пространства (то, что находится в центре программы). Сейчас у нас отображается информация о проекте (в разделе файлов выделен именно файл нашего проекта). Рассмотри основные два состояния:

1. Когда в разделе файлов левой панели у нас выбран любой файл с кодом, настройками проекта, ресурсами или просто папка (на самом деле всё, кроме файлов визуальных представлений о которых позже). В таком случае имеют место быть следующие разделы:
   - Инспектор свойств выбранного файла (здесь чаще оказывается нужным раздел принадлежности к проекту, особенно, если проект разбит на множество мелких подпроектов и не хочется, чтобы их код переплетался, либо наоборот был общим)



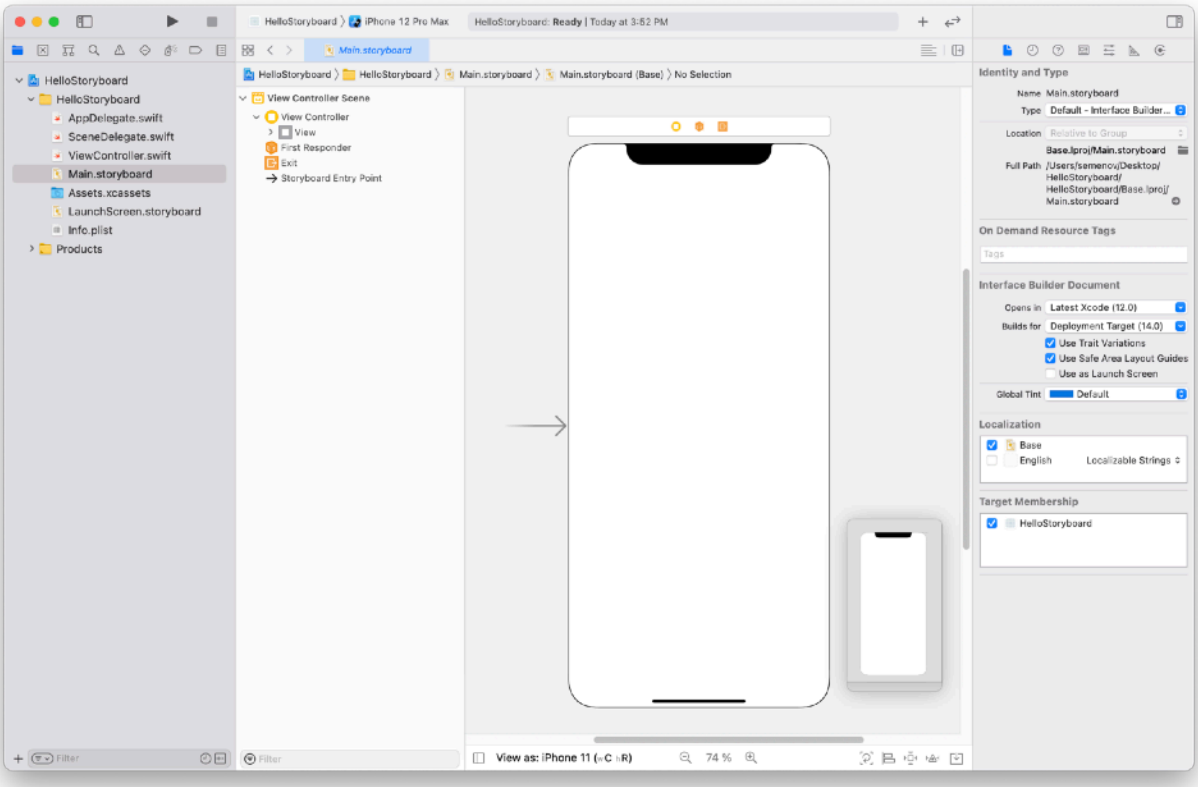
   - Инспектор всех изменений выбранного файла (при работе с git)
   - Инспектор справки (быстрый доступ в документацию, либо быстрая информация о выбранной команде, если находимся в файле с кодом)
   - Инспектор атрибутов (опционально, если есть, что настроить для текущего файла) Например для ресурсов можно указать для каких устройств они будут

использоваться (к примеру изображения большего разрешения для iPad и меньшего для iPhone)

2. Когда выбран файл представления (.storyboard или .xib):
   - Первые 3 раздела с тем же функционалом
   - Инспектор идентификации представления (каждое представление должно быть уникальным, иметь класс, к которому относится, иметь свой ID)
   - Инспектор атрибутов для настройки представления (цвет, стиль, отступы и др.)
   - Инспектор размера представления
   - Инспектор связей между элементами представления и кодом

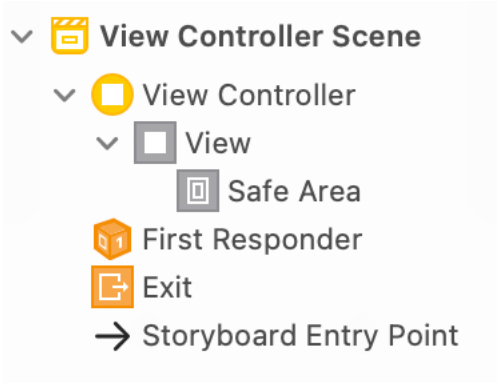
Между разделами правой панели можно также переключаться через cmd+option+порядковый номер, либо используя горячую комбинацию из настроек. Теперь давайте поговорим о том, что же такое storyboard.

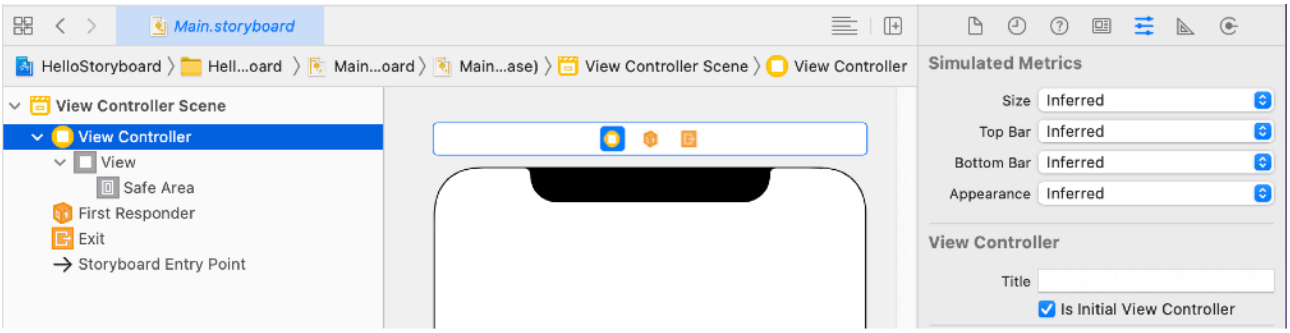
Storyboard файлы, созданные для нашего проекта по умолчанию это LaunchScreen.storyboard, который статичен и отвечает за то, что пользователь будет наблюдать, пока наше приложение загружается (обычно здесь размещают какое-то лого или название приложения, в Apple же рекомендуют размещать здесь что-то максимально похожее на главный экран приложения, чтобы пока приложение грузится, пользователь уже увидел как-бы его функционал, имитировав тем самым более быструю загрузку) и Main.storyboard, являющийся стандартным представление для нашего главного экрана. Давайте выберем этот файл, чтобы у нас открылось рабочее пространство сториборда.



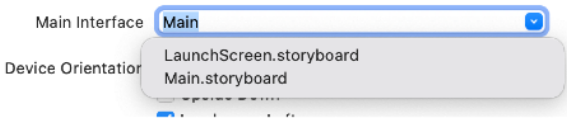
Идея сторибордов изначально в том, чтобы была возможность настроить внешний вид приложения с минимальным знанием кода. В открывшемся окне мы видим первый экран приложения, который пока что пустой. Далее мы сможем добавлять здесь различные кнопки, поля ввода, текст и так далее. Сейчас давайте обратим внимание на иерархию представлений (в левой части рабочего пространства). Есть контроллер сцены, который включает в себя несколько элементов (рассмотрим пока что часть из них), один из которых это обычный контроллер экрана (тот кто контролирует всё отображение текущего экрана, здесь

будет вся основная логика нашего приложения), содержащий в свою очередь корневую view (то, что конкретно будет отображаться на экране). Вообще идею разделения экрана на контроллер и вью стоит рассматривать как разделение на отдельные по смыслу части. В контроллере происходит вся логика нашего приложения, связанная с тем что показать, что делать, если будет нажата кнопка, как перейти на другой экран, и так далее. Вью же просто является визуальным шаблоном на экране, который может быть наполнен любым содержимым. Это та часть нашего приложения, которая видна пользователю (в отличии от контроллера). Также здесь есть такой компонент, как "Точка входа" в приложение (стрелка указывающая на экран слева). Без этой стрелки в проекте не будет назначен главный стартовый экран и при запуске всё просто будет черным. Назначить экран стартовым можно выбрав нужный контроллер (либо в иерархии, либо над самим экраном по первому значку) и, перейдя в инспектор атрибутов поставить галочку "Is Initial View Controller".





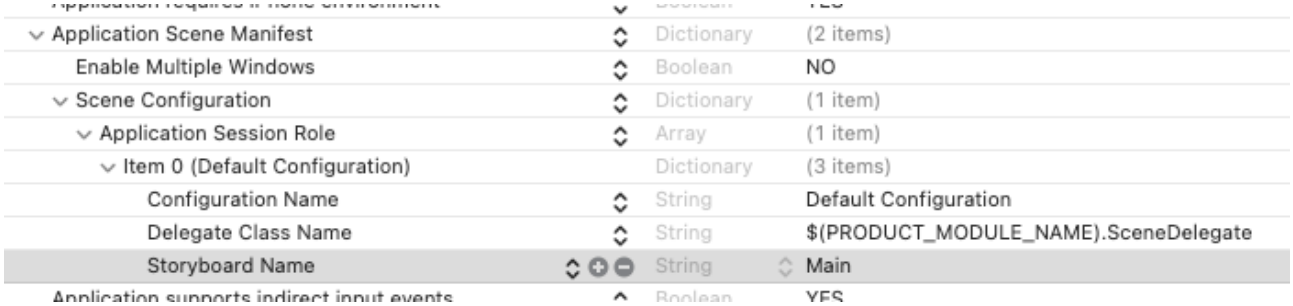
Файл Main.storyboard это обычный файл, который был создан в качестве шаблонного, так как мы выбрали создания шаблона приложения. В любой момент времени вы можете создать свой такой storyboard файл для создания нового экрана в вашем приложении. Хотя обычно разработчики создают все свои экраны внутри одного storyboard файла. Такой подход имеет место быть, но крайне нежелателен, потому как это подходит только для работы одним человеком на проекте, ведь любое изменение (на самом деле даже открытие storyboard файла для другой модели устройства) создает изменения которые будут отражены в системе git, но в тоже время будут совершенно не читаемы, так как представляют из себя XML файл. Как же приложение узнает о том, что нужно запустить именно Main файл и его точку входа в качестве стартового экрана. Всё просто, эту настройку можно было видеть в параметрах файла проекта, которая отображалась когда проект был только что создан.



Давайте теперь поговорим о файлах из которых состоит проект. Первое с чем мы познакомимся это файлы AppDelegate и SceneDelegate. Раньше, до iOS13 приложения создавались только с файлом делегата приложения, который можно было использовать для настройки поведения приложения, то есть через этот файл происходило взаимодействие операционной системы с нашим приложением. После iOS13 операционная система для планшета отделилась в отдельную систему iPadOS, где появилась поддержка нескольких окон приложений на одном экране устройства. В связи с этим появился файл делегата сцены. Сейчас по умолчанию

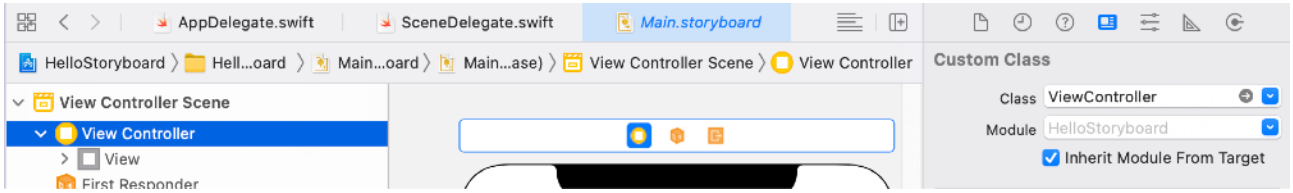
всегда создаются два этих файла. Даже если вы не собираетесь создавать приложение для iPad с поддержкой многооконного режима. В интернете даже есть информация о том, как удалить файл SceneDelegate из проекта, но на самом деле это не так уж и важно. Ваше приложение будет корректно работать и в случае полноэкранного представления на телефонах. По молчанию в этих файлах содержится множество методов, позволяющих отслеживать состояние приложения, а также подробные комментарии о том для чего нужны эти методы. Если избавиться от всего лишнего, что необходимо для работы с сторибордом, то останется лишь свойство window в файле SceneDelegate (потому что это свойство отвечает за наличие в вашем приложении как такового визуального представления на экране) и единственный метод, связывающий файл делегата приложения со сценой по умолчанию (информацию об этой сцене можно найти в файле настроек приложения Info.plist).

```
@main
class AppDelegate: UIResponder, UIApplicationDelegate {

    // MARK: UISceneSession Lifecycle
    func application(_ application: UIApplication, configurationForConnecting
        connectingSceneSession: UISceneSession, options: UIScene.ConnectionOptions)
        -> UISceneConfiguration {
        return UISceneConfiguration(name: "Default Configuration", sessionRole:
            connectingSceneSession.role)
    }
}
```

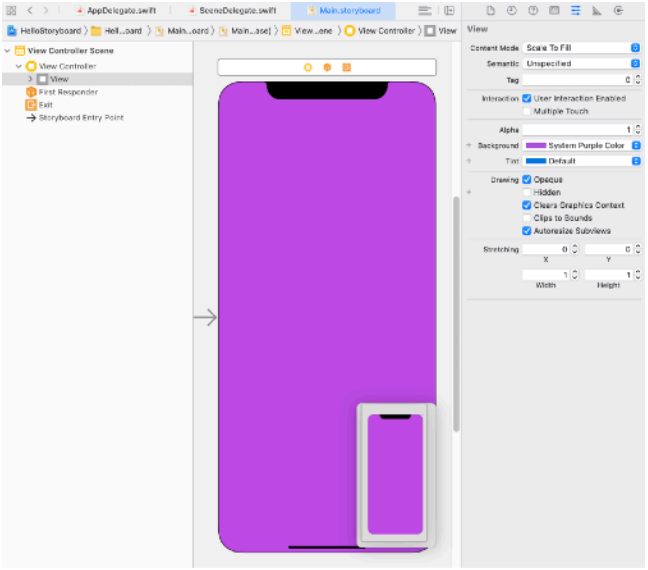


Так что в вашем приложении нет никакой магии, все предельно просто. Если есть какой-то функционал, значит непременно где-то есть его упоминание для корректной работы. Отмечу также, что часто разработчики не задумываются об этих файлах и оставляют весь шаблонный код здесь вместе с комментариями. Я же настоятельно рекомендую не плодить в своем проекте лишний мусор и избавляться от всего, что не используется. Если вдруг вам понадобится какой-то из методов делегата в будущем, это можно решить как минимум созданием пустого чистого проекта (чтобы скопировать код), либо загуглив, а то и просто обратившись к документации, там все предельно подробно описано, но разве что на английском.

Далее давайте обсудим то, как мы можем связать наше представление сториборда с кодом. На самом деле в проекте уже есть файл ViewController.swift, внутри которого определен стандартный класс для базового контроллера. Единственное, что там пока что пусто. Если мы снова вернемся в файл Main.storyboard, выберем контроллер и откроем раздел инспектора идентификации (иконка в виде карточки), мы увидим то же название класса в поле класса экрана



сториборда. Таким образом класс в коде и представление в сториборде связаны и работают как единое целое. В любой момент времени вы можете создать свой класс или свой файл сториборда и связать их, указав нужное имя класса для идентификации экрана сториборда. Только не забудьте, что данный класс должен быть наследником от стандартного класса UIViewController, в котором определена работа экрана, а также у вас в файле с кодом должна быть импортирована библиотека UIKit.



Давайте попробуем, наконец, запустить наш проект, но чтобы быть уверенными, что запустилось именно то представление, которое нам необходимо, поменяем фон экрана с белого на любой другой. Для этого в файле Main.storyboard необходимо выбрать в иерархии корневую view и в инспекторе атрибутов указать желаемый цвет. После чего, предварительно выбрав любую модель симулятора или устройство, можно нажать кнопку запуска в верхней панели или комбинацию cmd+R. Сначала проект соберется, после чего запустится и установится симулятор (если это первый запуск) и отобразится приложение с тем цветом фона, который был выбран.

Не смотря на то, что больше внимания я буду уделять визуальной настройке экрана приложения через storyboard, здесь есть не все возможные настройки и что-то удобнее будет выполнить из кода. Например, как мы можем задать цвет нашего экрана из кода? Учитывая, что у нас уже есть файл контроллера, внутри которого находится корневая view, мы можем в этом файле менять ее параметры точно так же, как делали это в инспекторе атрибутов. Давайте попробуем. В фале ViewController.swift удалим комментарий внутри метода viewDidLoad и вместо него поменяем значение свойства backgroundColor у корневого вью, например так:

```
view.backgroundColor = .systemBlue
```

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        view.backgroundColor = .systemBlue
    }
}
```

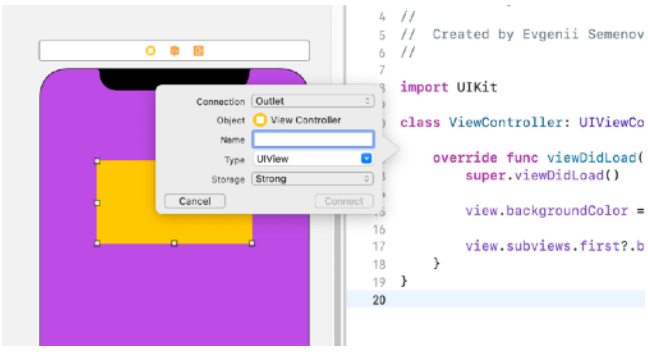
Здесь я указываю в качестве фона экрана значение системного синего цвета, являющегося статичным свойством класса UIColor. Это класс, отвечающий за представления цвета в библиотеке UIKit. Для данного класса есть множество стандартных цветов, а также возможность вызвать конструктор с заданием любых компонент цвета, для получения своего собственного цвета). Подробнее на этом я не буду останавливаться, так как это просто класс, от которого можно задать экземпляр со своими характеристиками, а все преобразование в цвет происходит внутри нашего приложения и напрямую этот функционал нам не доступен. Если мы снова запустим приложение, то на этот раз увидим, что цвет главного экрана поменялся. Однако в сториборде цвет остался неизменным, как же так происходит. Как понять, какие параметры экрана будут отображены? При загрузке экрана, до его появления у нас прежде всего загружается вся информация, находящаяся в файле сториборда, а уже после вызываются методы класса, описанные в коде. Получается, что экран приложения

сначала был установлен в пурпурный, а после перекрашен в системный синий. Причем все это произошло еще до отображения view (viewDidLoad срабатывает когда загружена вся информация, необходимая для отображения корневой view на экране, подробнее об этом и других методах будет в следующих уроках) на экране симулятора, поэтому мы не увидели мерцания или быстрого переключения цвета.

Теперь давайте попробуем добавить на экран приложения ещё одну view и поменять ее цвет также, как мы сделали это с корневой view. Для того, чтобы добавить что-то в сториборде, надо открыть библиотеку компонентов (cmd+shift+L или иконка "+" в верхней панели) и выбрать там необходимый компонент. Так как нам нужна view в поиске можно написать UIView (если написать просто view, будет предложено много других компонент, содержащих слово view в названии). Теперь просто берем view из списка и перетаскиваем на экран. Можно задать ему любой размер, потянув за угол или сторону и указать любой цвет в инспекторе атрибутов. Если теперь запустить проект, мы увидим, что на экране добавилась область другого цвета. Теперь давайте обсудим то, как мы можем поменять цвет этой view из кода. Несмотря на то, что наше представление в сториборде связано с классом ViewController у нас нет сейчас прямой ссылки на новую view. Все, что мы можем сделать сейчас, это взять первую дочернюю view у корневой и попробовать задать ей цвет.

```
view.subviews.first?.backgroundColor = .green
```

Действительно, в таком случае мы попали в нужную view, но что, если их у нас будет много. В таком случае проще всего создать связь данного представления с кодом напрямую (так называемый outlet). Для этого мы можем создать специально описанное свойство для класса контроллера и потом связать его с view в сториборде, а можем сделать так, чтобы Xcode создал это свойство за нас. Рассмотрим второй способ, так как первый после будет очевиден. Чтобы связать код и представление сториборда, надо одновременно на экране открыть два этих файла. Сделать это можно открыв один из файлов, после чего щелкнув по другому файлу с зажатой клавишей option. Рабочее пространство по середине экрана разделится на 2 области. Далее зажимаем клавишу control и перетягиваем нашу view в начало класса ViewControler (сделать это можно либо из иерархии представлений, либо прямо с экрана сториборда). Отобразится окно создания связи, где видно, что это будет outlet (свойство) для выбранного экрана. Все, что от нас требуется, это задать данному свойству имя (например myView сейчас будет достаточно) и нажать "Connect". Тип объекта и ссылки можно оставить по умолчанию. После в коде появится код, описывающий дополнительное свойство класса (со специальной пометкой IBOutlet, здесь IB - interface builder, то есть то самое представление сториборда для построения экрана). Теперь не составит труда поменять свойство данной view, обратившись к ней напрямую.



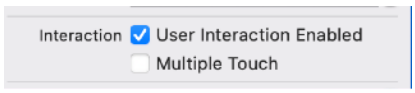

```
myView.backgroundColor = .green
```

Пару слов о том, почему тип ссылки мы не меняли (по умолчанию он weak). Это связано с тем, что раньше, при переходе между экранами в приложении визуальное представление прошлого экрана выгружалось из памяти (потому что в устройствах было мало оперативной памяти). И чтобы экран успешно выгрузился, необходимо было, чтобы на него более никто не ссылался, после ухода с него. Что успешно решалось weak ссылкой для всех свойств класса контроллера на элементы представления. Сейчас же такой необходимости нет, поэтому можно все свойства объявлять как strong, но учтите, что ваш экран всегда будет висеть в памяти, даже если вы ушли на другой.

Ранее я писал о том, что в сториборде возможно настроить не все, что нам бы хотелось, но до этого приводил примеры со сменой цвета из кода, что можно было сделать и без помощи кода. Давайте далее рассмотрим одну из задач, которая не может быть решена с помощью сториборда. Что, если при старте приложения мы хотим задать для view не определенный цвет, а случайный? Здесь не обойтись без кода. Можно закрыть представление сториборда, чтобы вернуться в полноэкранный режим. Зададим в классе массив из нескольких цветов и при старте приложения будем выбирать случайный цвет для маленькой view.

```
private let randomColors: [UIColor] =
[.red, .green, .blue, .brown, .yellow, .cyan, .magenta, .orange]

override func viewDidLoad() {
    super.viewDidLoad()

    view.backgroundColor = .systemBlue
    myView.backgroundColor = randomColors.randomElement()
}
```

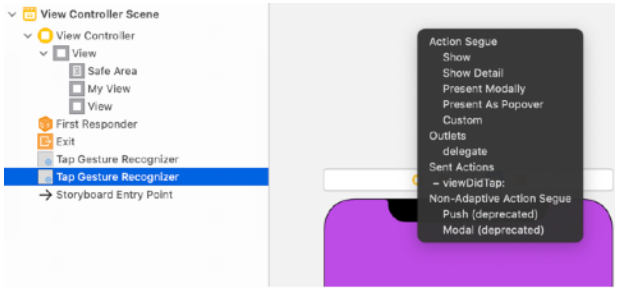
Вот так вот легко мы добились чего хотели. Поговорим теперь про взаимодействие с нашим приложением. Пока мы просто отображаем какую-то информацию на экране это не столь интересно. Хочется добавить небольшого интерактива. Давайте по нажатию на маленькую view меня ее цвет на другой случайный цвет. Для этого поговорим о жестах, доступных нам с библиотекой UIKit. В приложение мы можем добавить поддержку совершенно различных жестов, более того, которые элементы экрана (например кнопка) уже поддерживают некоторые жесты сами по себе. Но view является обычным отображением, поэтому по умолчанию никак не взаимодействует с пользователем. Точнее взаимодействует, но мы никак не обрабатываем это взаимодействие. Вернёмся в сториборд и выделим myView. В разделе инспектора атрибутов здесь есть пункт "Взаимодействие" и здесь по умолчанию уже стоит галочка взаимодействовать с пользователем. Не хватает лишь какого-то механизма, позволяющего реагировать на нажатия пользователя. Этот механизм не что иное как распознавание жестов. Чтобы добавить какой-либо жест элементу на экране снова идем в библиотеку элементов, ищем там UITapGestureRecognizer и перетаскиваем его на myView в сториборде. Отлично, теперь мы отслеживаем на данной view такой жест, как нажатие, все что осталось сделать, это вызывать некоторый метод класса контроллера, когда произойдет нажатие. Снова открываем файл с кодом с зажатой клавишей option, а после, с зажатой клавишей control перетаскиваем жест в класс ViewController (методы лучше располагать в конце класса). В качестве типа соединения (первый параметр) выбираем Action (действие), потому что нас не будет инетресовать жест как таковой, нас будет интересовать метод, который сработает



при нажатии. Далее задаем имя, например viewDidTap и тип обработчика (здесь лучше выбрать жест, чем просто оставить Any, тогда в случае чего, нам не нужно будет лишний раз кастить обработчик до нужного типа), после чего нажимаем "Connect". Можно закрыть окно сториборда, в коде видим добавившийся метод. Также обратите внимание, что напротив (слева) всех методов и свойств, добавленных через сториборд у нас есть серые точки, отображающие наличие этой связи. Если вдруг в какой-то момент времени вы удалите что-то из сториборда, но не удалите связь с этим объектом из кода (точка будет пустой), ваше приложение будет падать, будьте внимательны при работе со всеми связями вашего кода со сторибордом. Осталось только скопировать команду задания случайного цвета для myView и вставить в новый метод обработки жеста.

```
@IBAction func viewDidTap(_ sender: UITapGestureRecognizer) {
    myView.backgroundColor = randomColors.randomElement()
}
```

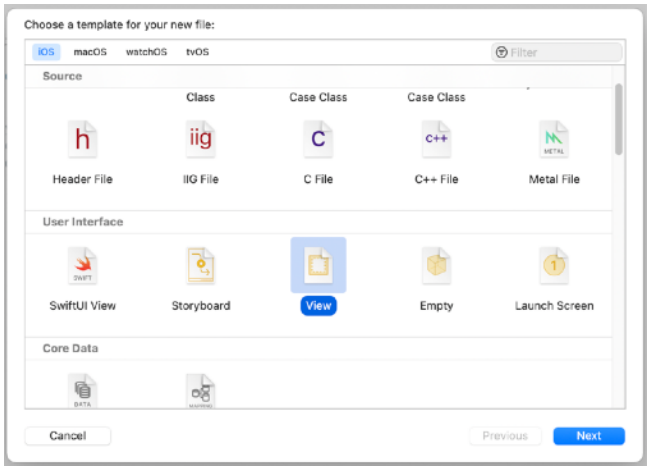
Теперь давайте добавим еще одну view на сториборд и жест нажатия для нее. Добавляем view из бибилиотеки на экран и добавляем жест из той же бибилиотеки на эту view. Что если мы хотим чтобы её цвет тоже менялся при нажатии. Если мы поступим таким же образом, как и с myView, то нам нужно задать свойство с данной view в классе контроллера и создать Action с обработкой нажатия, где выставить цвет для данный view в случайный из массива randomColors. Но на самом деле можно проще. Нам не нужно знать про саму view в классе, потому что когда у нас вызывается Action, в качестве аргумента мы получаем жест, у которого хранится информация в том числе о том, е чему был применен данный жест.Так что все, что мы сделаем, это свяжем новый жест с уже имеющимся методом. Причем нам не обязательно открывать для этого режим разделения экрана с кодом, потому что код уже есть. Это в том числе второй способ создания связи, о котором я писал выше. Когда мы вручную создаете свойство или метод со специальным атрибутом "@IB" вначале. Просто берем и перетаскиваем жест на контроллер с зажатой клавишей control (ViewController в иерархии представлений, либо первая иконка с желтым кругом и белым квадратом в нем), а в появившемся списке выбираем интересующий нас метод viewDidTap. Если сейчас запустить проект, то при нажатии на любую маленькую view будет меняться цвет myView. Это не совсем то, что мы ожидали, но это именно то, что написано у нас в коде метода по обработке нажатия. Для того, чтобы менять цвет именно той view, которая была нажата, необходимо получить ее из жеста, приходящего в метод в качестве аргумента. Заменяем код метода на новый:



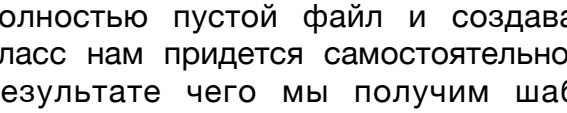

```
sender.view?.backgroundColor = randomColors.randomElement()
```

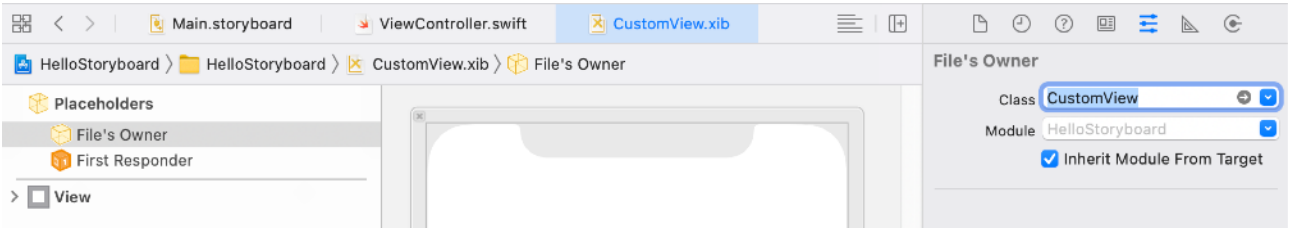
Таким образом та view на которую мы нажали, будет передана в качестве свойства с одноименным именем у объекта жеста. Теперь все работает как надо, однако у нас есть две совершенно одинаковые view, которые выполняют одно и то же, давайте попробуем их переиспользовать. Вообще в Xcode есть возможность создать отдельный storyboard файл, но не для всего экрана, а для отдельного его элеменат, например некоторой view. Такие файлы называются xib файлами, но

почему-то для обычных view они создаются крайне сложно. Рассмотрим пример создания xib файла для view в несколько шагов.



Для начала добавим файл с кодом для класса view. Для этого либо в интерфейре файлов и папок нажимаем правую кнопку мыши и выбираем New File, либо нажимаем комбинацию клавиш cmd+N. У нас етсь две возможности создать файл с кодом, это обычный SwiftFile. В таком случае мы получим полностью пустой файл и создавать класс нам придется самостоятельно, либо можно выбрать CocoaTouchClass, в результате чего мы получим шаблонный класс с уже импортированной библиотекой и некоторыми стандартными свойствами или методами. Давайте выберем второй вариант. Сначала укажем в качестве родительского класса UIView (вверху выпадающего списка), после чего укажем название класса для наследника, в моем случае это просто CustomView. В качестве языка оставляем Swift, а также обратим внимание на то, что здесь есть возможность поставить галочку "также создать xib файл", но почему-то для обычной view (прямого наследника от UIView) эта опция недоступна. Это та трудность с которой нам придется сегодня столкнуться и разобраться. Позже в курсе мы рассмотрим как можно будет легко и просто создавать xib файлы для некоторый вариантов view, но для UIView сейчас нам нужно будет проделать много всего собственноручно. Итак, нажимаем "Далее" и сохраняем новый файл в папку нашего проекта. Следующим шагом снова идем создавать файл (можно cmd+N), но на этот раз нужно прокрутить чуть ниже и найти View в разделе "User Interface" (то есть UI элемент). Здесь необходимо задать точно такое же имя, какое было задано классу view (в моем случае это CustomView) и снова сохранить файл в папку проекта.

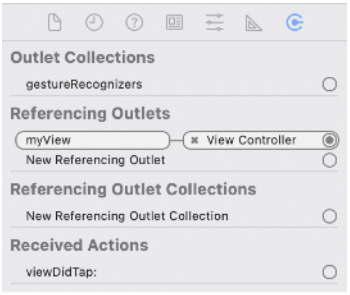




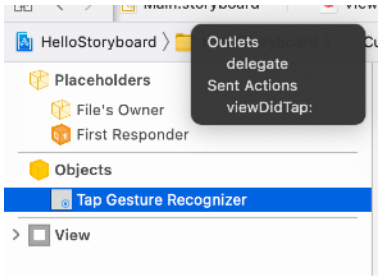
Теперь откроем xib файл и прежде всего зададим связь между классом и xib файлом. Сделать это можно выбрав "File's Owner" в левой панели и указав необходимый класс в инспекторе атрибутов в правой панели. Также в качестве размера для самой view в инспекторе атрибутов укажем "Freeform" и сделаем размер view немного меньше чем сейчас (во весь экран).

Давайте перейдем обратно в Main.storyboard и укажем для каждой маленькой view класс CustomView в качестве основного класса. Теперь нам не нужны жесты в контроллере и их обработка в классе контроллера, поэтому перенесем код в класс CustomView.

```
class CustomView: UIView {

    private let randomColors: [UIColor] =
[.red, .green, .blue, .brown, .yellow, .cyan, .magenta, .orange]

    @IBAction func viewDidTap(_ sender: UITapGestureRecognizer) {
        sender.view?.backgroundColor = randomColors.randomElement()
    }
}
```



В классе контроллера мы можем теперь удалить все, и свойство myView и массив цветов, и Action со сменой цвета. Оставьте только метод viewDidLoad, в котором будет устанавливаться цвет корневой view. В сториборде можно удалить жесты, для этого необходимо выделить каждый и нажать клавишу "backspace" ("delete"). Также надо не забыть избавиться от связи myView с кодом, поэтому выделяем

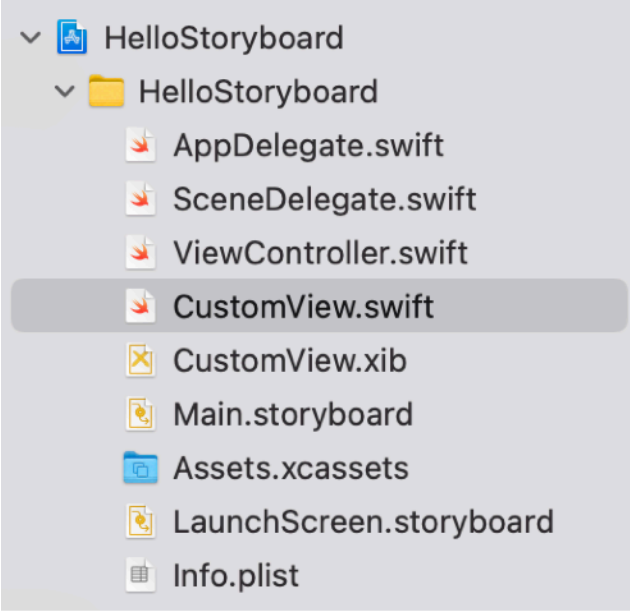


данную вью и в инспекторе связей удаляем связь с контроллером, нажатием на маленький крестик перед именем контроллера. Вернемся в xib файл CustomView, добавим из библиотеки жест для этой view, после чего перетянем его с зажатой клавишей control на элемент "File's Owner" в иерархии объектов слева. Выберем здесь метод viewDidTap, который мы перенесли в класс CustomView. Давайте попробуем запустить проект и проверить, что мы все перенесли корректно (нет). На деле окажется, что по нажатию цвет view теперь не меняется. Где же мы ошиблись? Если бы нам дали поставить галочку "также создать xib файл", то наши view, когда загружались вместе с главным экраном знали бы, что им необходимо загрузить свое представление в xib файле. Точно также как в настройках нашего проекта прописано, что загружать проект нужно с файла Main.storyboard, здесь же указан и класс, с которым мы будем работать и точка входа. Но со view ситуация немного другая. В сториборде главного экрана мы указали, что каждая view должна быть связана с классов CustomView и в xib файле мы тоже указали, что этот класс будет "владельцем" нашего xib файла, то есть будет им управлять. Однако мы нигде не указали для самого класса, что он должен хоть что-то знать про xib файл. Сейчас он его даже не замечает, будто бы xib файла и нет вовсе. Сами view знают только про файл класса и мы можем убедиться в этом, если, например, зададим для наших view начальный цвет. Сделать это можно в определении инициализатора CustomView. Например так:

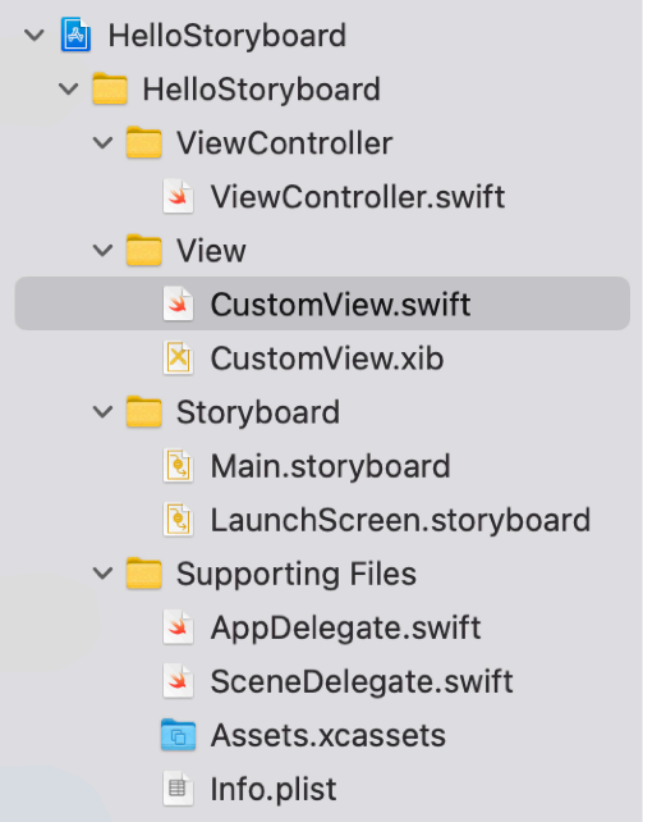
```
required init?(coder: NSCoder) {
    super.init(coder: coder)

    backgroundColor = .cyan
}
```

Если теперь запустить приложение, то цвет маленьких view действительно изменится. Далее рассмотрим как же мы можем потребовать у класса CustomView загрузить всю информацию, содержащуюся в xib файле. Прежде чем мы это сделаем, давайте наведем в проекте порядок. Для всех сущностей проекта создадим отдельную группу (папку).

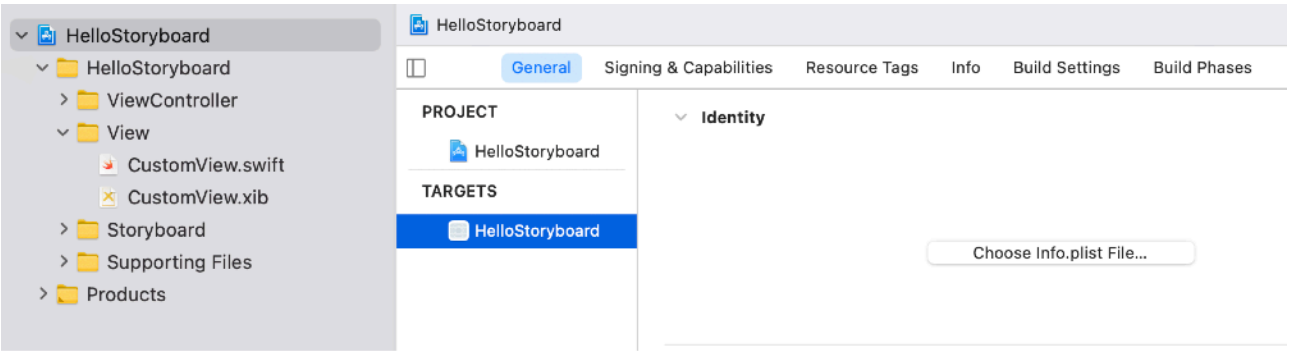
Было:



Стало:



У Xcode есть одна интересная особенность, он каждый раз ломается, когда происходит какой-то перенос или переименование файлов. Это старая проблема, которую возможно в будущем починят, но пока что, если мы попробуем запустить проект, получим ошибку, связанную с тем, что у нас в проекте отсутствует файл настроек (потому что мы перенесли его в папку Supporting Files). Но просто так его указать не получится, нужно перезапустить проект (нажав cmd+Q в Xcode). Теперь, если перезапустить проект и выбрать его в навигаторе файлов, нам предложат выбрать расположение Info.plist файла. После чего проект снова будет собираться и запускаться.



Теперь рассмотрим то, как мы можем связать класс view и xib-файл в обратном порядке. Для этого создадим расширение для UIView класса, в котором добавим такую возможность для любой view, на будущее. Либо, данное расширение вы также можете использовать в своих проектах. Прежде всего создаю новую группу (папку) Extension, а внутри swift файл с именем UIView+Extension. Назвать данный файл можно как угодно, здесь нет какого-то общепризнанного правила, просто удобнее, если в названии файла содержится информация о том, что это за файл. Другой пример более короткого названия просто +UIView, так как итак понятно, что данный файл находится внутри папки "расширение" и здесь будет добавлено что-то для класса UIView. Файл будет

содержать расширение с всего лишь одним методом, который будет загружать настрйоки из xib файла:

```
import UIKit

extension UIView {

    func getViewFromXib() -> UIView? {
        let className = String(describing: Self.self)
        let anyNib = Bundle.main.loadNibNamed(
            className,
            owner: self,
            options: nil
        )?.first
        guard let customView = anyNib as? UIView else { return nil }
        return customView
    }
}
```

Разберем принцип работы данного метода. Сначала мы получаем имя того класса для которого будем загружать xib. Здесь для этого используется стандартная для этого конструкция по обращению к типу текущего класса (Self.self, что в нашем случае будет аналогично CustomView.self, то есть сам по себе класс CustomView), после чего мы преобразуем данный класс в строку, через String(describing:). На выходе в константе className у нас будет содержаться строка с именем того класса, в котором был вызван данный метод (в нашем случае "CustomView"). Далее мы загружаем все xib файлы в нашем проекте (Bundle.main - это основной раздел нашего проекта, который будет сформирован при сборке проекта в исходный исполняемый файл, туда будут скопированы все ресурсы и зависимости), которые содержат такое имя и для который в качестве владельца указан текущий класс (в качестве File's Owner мы как раз установили класс CustomView). В названии метода загрузки используется слово "nib", как более старое и общее название для файлов interface builder, где N - NeXT (система, разрабатываемая Джобсом отдельно и в последствии влитая в macOS), а X - macOS X (новое и последнее поколение систем), поэтому можно считать это одним и тем же. В качестве опций мы ничего не указываем, а в конце используем свойство first, потому что на выходе мы получаем массив загруженных объектов, если будут совпадения (у нас файл один, так что можно смело брать первый попавшийся. Обратите внимание, что здесь происходит работа с опционалами, потому как мы могли забыть указать управляющий класс для xib файла, либо указать отличное имя. В таком случае загрузить содержимое xib файла у нас не получится. Осталось только понять, что мы только что загрузили. Если это объект, который можно рассматривать как экземпляр класса UIView, мы получаем данный объект и возвращаем его. Вернёмся в класс CustomView и добавим загружаемую их xib view в качестве приватного ленивого свойства, а в методе илициализатора из coder (который вызывается при создании объекта не через код, а через storyboard представление) сразу после вызова инициализатора родителя инициализируем данное свойство и добавим эту xib-view в качестве дочерней view, теперь мы будем работать с ней, будто это и есть наша основная view. Для нее мы теперь зададим свет фона, а также, если в xib файле она имеет отличный размер, мы можем добавить метод layoutSubviews(), срабатывающий каждый раз при перерисовки данной view на экране, и задать для xib-view такой же размер, как оригинальная view. В итоге должен получиться такой код:

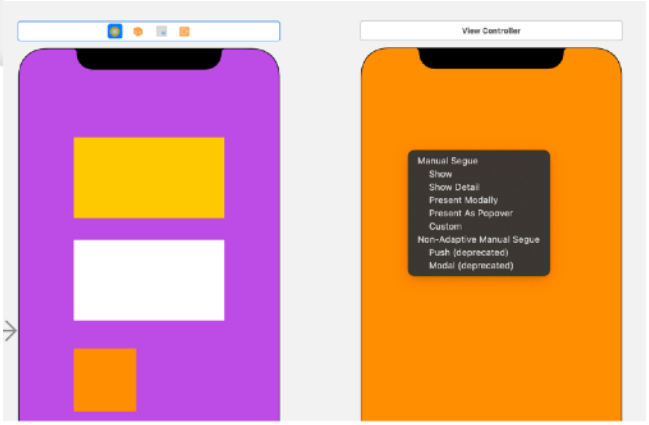
```
private lazy var xibView: UIView = getViewFromXib() ?? UIView()

required init?(coder: NSCoder) {
    super.init(coder: coder)

    addSubview(xibView)

    xibView.backgroundColor = .cyan
}

override func layoutSubviews() {
    super.layoutSubviews()

    xibView.frame = self.bounds
}
```

Если вы забыли, что такое ленивое свойство, то это то свойство, которое будет вычислено лишь при первом обращении к нему. А так как результат метода у нас опциональный, если у нас не получится загрузить xib файл, мы просто подставим пустую view, чтобы программа не упала. О том, как здесь задается размер, используя свойства frame и bounds у view мы поговорим чуть дальше в курсе, но идея данной строчки в том, что мы должны задать для xib-view такой же размер, как у текущей view и должны расположить ее начиная от самого верхнего левого угла (с нулевым смещением) текущей view, чтобы они полностью совпали. Мы неспроста ориентируемся именно на левый верхний угол, потому как в Xcode, при создании приложений под iOS координатная сетка начинается от левого верхнего угла устройства. Ось OX направлена вправо, а ось OY направлена вниз. Если теперь запустить проект, мы увидим все те же view с заданным по умолчанию цветом, но при нажатии на них, на этот раз, они будут изменять свой цвет на случайный, указанный в массиве randomColors класса CustomView. Вся прелесть здесь в том, что мы просто добавили эти view в файл сториборда и указали в качестве класса для них класс CustomView. Вся логика смены цвета и начального цвета у нас описана лишь один раз именно в этом классе, потому что именно к нему она и имеет отношение, тогда как раньше у нас сам контроллер отвечал за раскрашивание свой view (вспоминаем SOLID принципы).

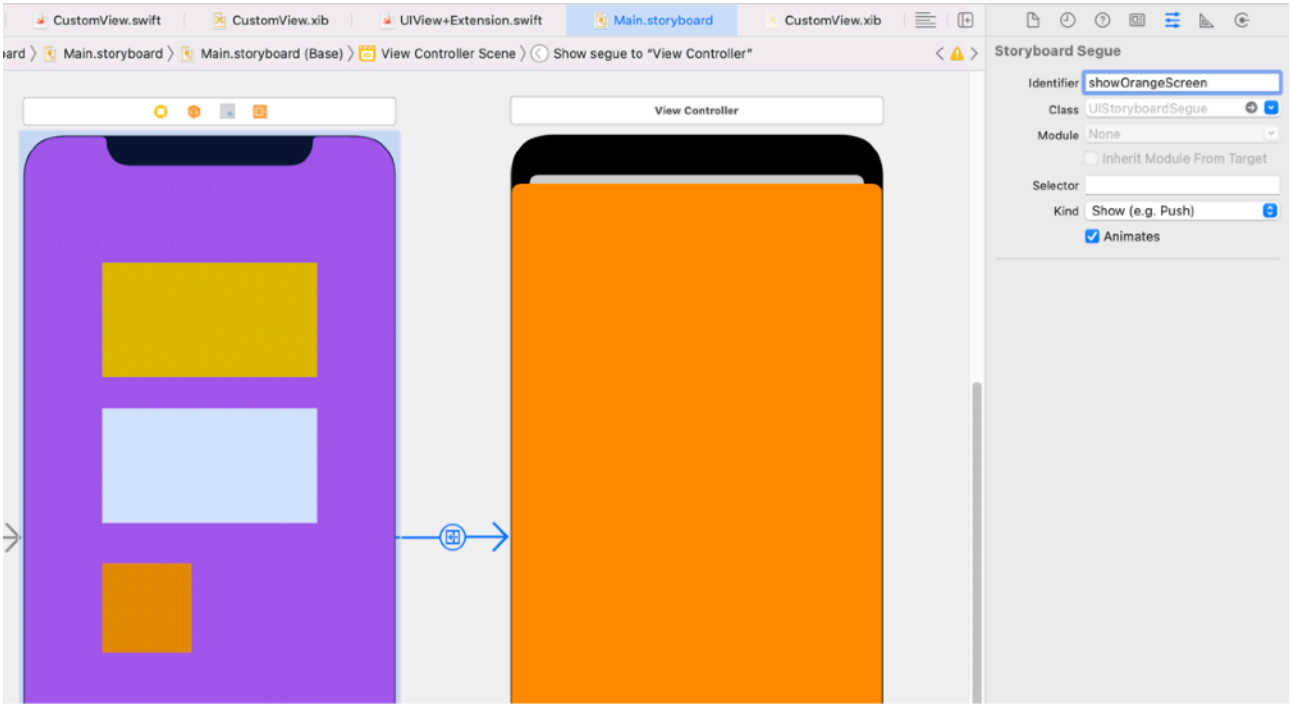
Последнее, что мы рассмотрим это создание дополнительных экранов приложения и простую навигацию между ними. Для этого вернемся в Main.storyboard и из библиотеки элементов вытащим ViewController, расположив его рядом с уже существующим главным экраном. Пока что не будем использовать какие-то готовые UI элементы (рассмотрим их в следующих уроках), поэтому сделаем открытие нового экрана при по жесту нажатия на некоторую view. Добавляем view на экран, добавляем для нее жест UITapGestureRecognizer и покрасим эту view и новый экран одинаковым цветом.

Теперь надо создать метод, который будет вызываться по нажатию на оранжевую view. Как раньше

открываем рядом с текущим файлом (зажав option и щелкнув на swift файл контроллера) и перетаскиваем в конец класса жест (предварительно зажав control). Я назову его orangeViewDidTap. Чтобы нам сделать переход между экранами, нам нужно создать возможность этого перехода в сториборде. Для этого нужно с зажатым control перетащить первый контроллер на второй (либо в иерархии, либо прямо на экранах). В появившемся окне



достаточно выбрать первый вариант "Show", потому что мы просто хотим отобразить экран. Если все сделано верно, между экранами появится стрелка с навигацией (segue или сегвей). А второй экран получил внешний вид модального экрана (отображаемого не во весь экран), мы исправим это позже, пока оставим как есть. Чтобы использовать segue в коде, ему нужно присвоить уникальный идентификатор. Поэтому щелкаем на данный переход и в



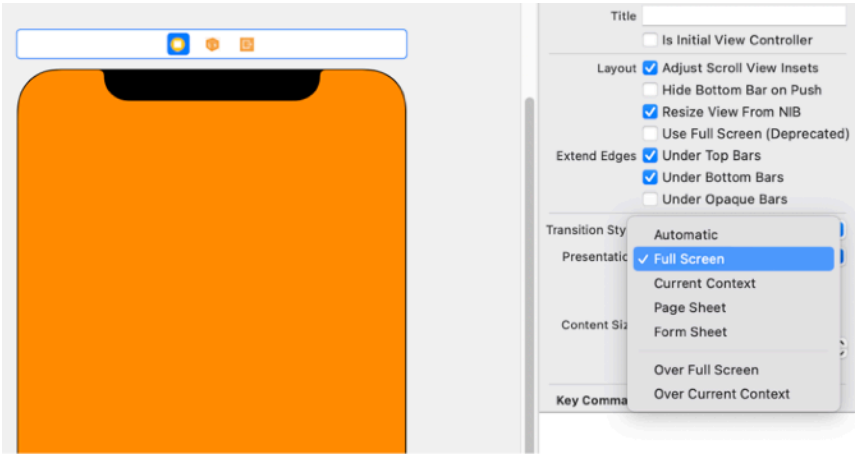
инспекторе атрибутов указываем любой идентификатор, лучше осмысленный. В моем случае это будет "showOrangeScreen". Теперь скопируем это название и перейдем в класс контроллера, чтобы сохранить данный идентификатор в коде.

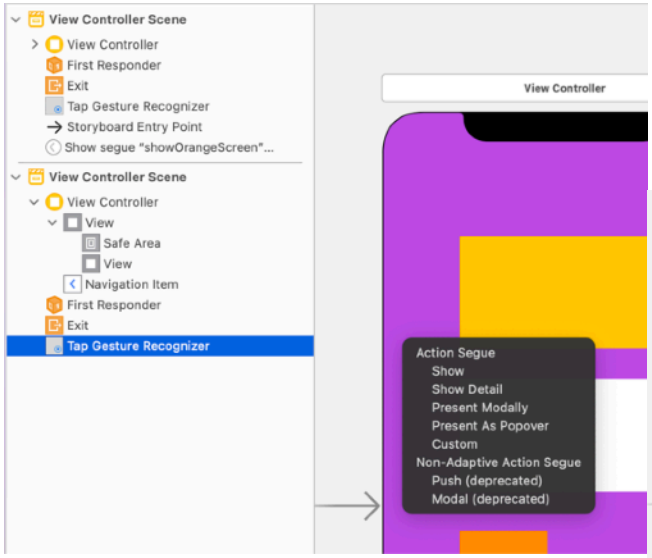
```
private let showOrangeScreenSegueID = "showOrangeScreen"
```

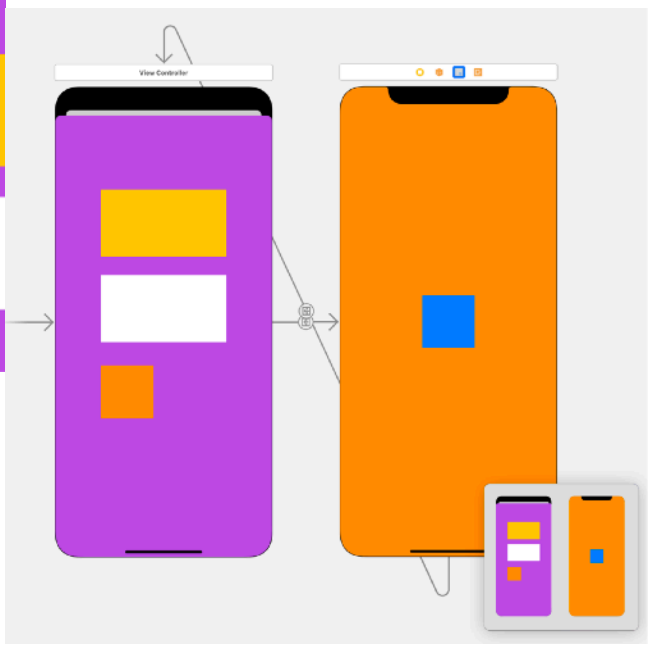
Чтобы все заработало нам осталось только вызвать данный переход при нажатии на оранжевую view. В методе обработки жеста нажатия добавляем вызов метода performSegue контроллера, где нужно указать тот самый идентификатор и того, кто вызывает данный переход, в нашем случае можно указать кого угодно, я укажу сам контроллер.

```
@IBAction func orangeViewDidTap(_ sender: UITapGestureRecognizer) {
    performSegue(withIdentifier: showOrangeScreenSegueID, sender: self)
}
```

Запустим приложение и нажмем на оранжевую view. У нас снизу всплывет оранжевый контроллер. Чтобы уйти с него, достаточно просто смахнуть его вниз. Вот мы и организовали первый переход между экранами. Для этого мы создали сегвей для этого перехода, указав откуда и куда будет переход, дали этому переходу уникальное имя и просто вызвали в нужный момент. Давайте вернемся в сториборд и зададим второму экрану полноэкранное представление. Для этого выделяем сам контроллер (не view, а именно контроллер) и в инспекторе атрибутов выбираем вариант презентации "Full screen". Если мы снова запустим проект и нажмем на маленькую оранжевую view, новый экран откроется во весь экран и выйти с него просто смахнув его вниз не получится, нужно сделать обратный переход. Но если мы сделаем как раньше, только перетянув сегвей уже со второго экрана на первый мы не вернемся назад, а лишь заново создадим первый экран и поставим его



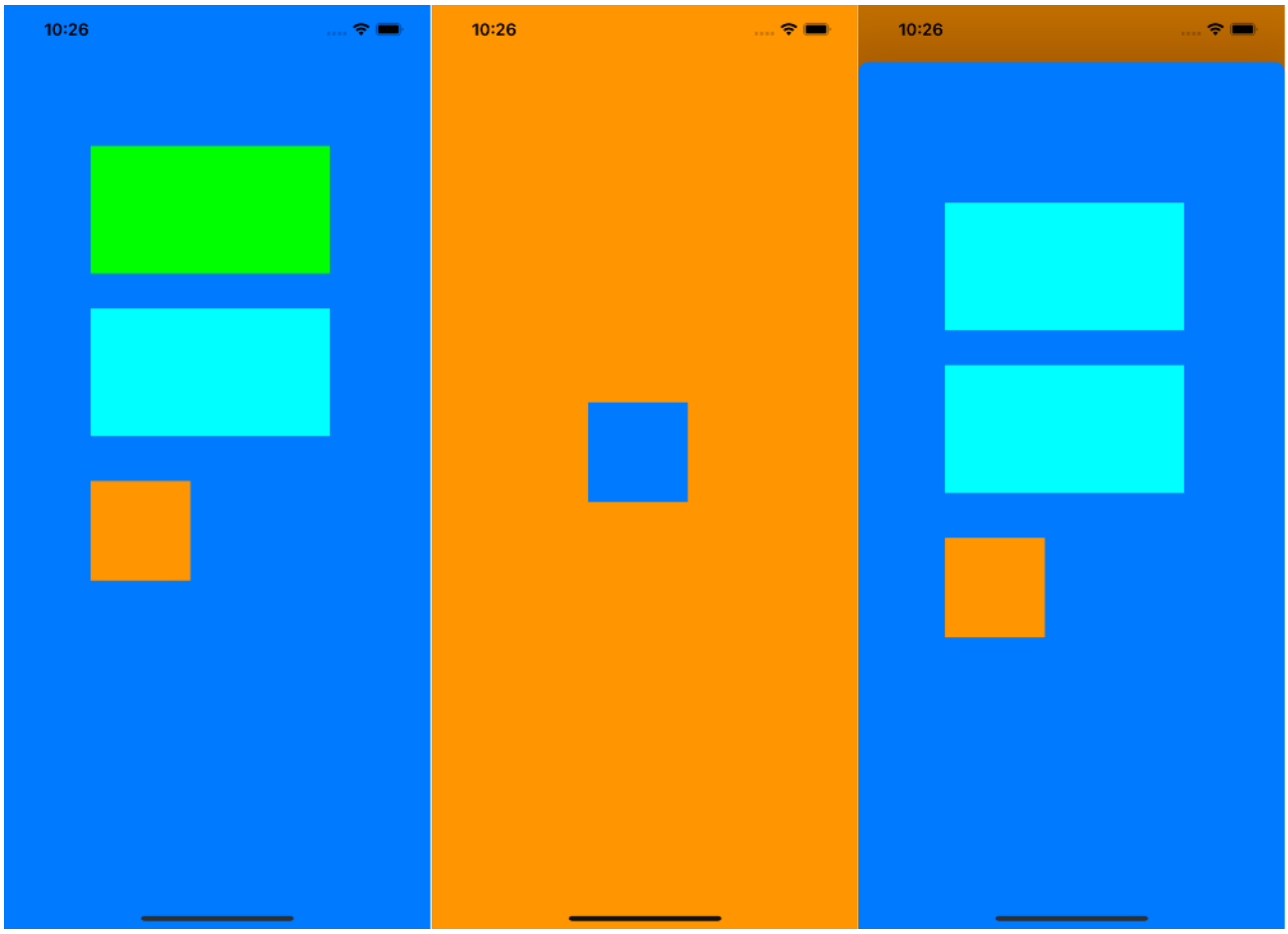




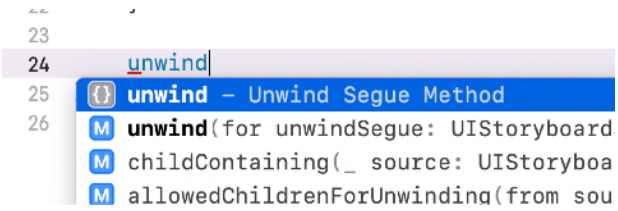
уже в качестве третьего по счету экрана и так далее, если мы будем продолжать переходы. Давайте убедимся в этом. Для этого на второй (оранжевый) экран поместим view, добавим для нее обработчик жестов и, так как мы не создавали отдельного класса

контроллера для этого экрана (то есть у нас нет возможности управлять им из кода), мы можем напрямую создать переход из этого жеста на первый экран. В таком случае перетянем жест view второго экрана на первый контроллер с зажатым control и в списке выберем "Show" для создания простого перехода. Получим зацикленную схему перехода между экранами.

Сейчас запустим проект, нажмем на маленькие view несколько раз для того, чтобы они поменяли свой цвет, после чего перейдем на оранжевый экран. Что теперь случится, если мы попробуем вернуться на синий экран нажав на синюю view? У нас поверх откроется первый экран с заново созданными маленькими view, которые имеют стандартный цвет. Мы можем так продолжать долго, пока память нашего приложения не закончится. Но это не то, что мы хотели получить. Наша задача была вернуться на предыдущий экран. Рассмотрим, как это сделать правильно, не пересоздавая экраны лишний раз.

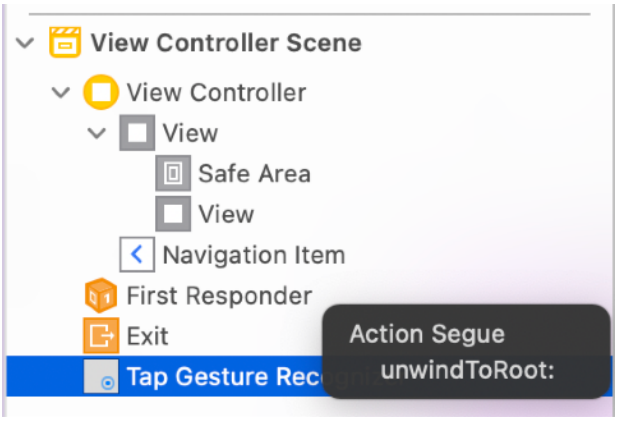


Во-первых давайте удалим тот сегвей, который добавили последним (для перехода со второго экрана на первый), мы сделаем его немного иначе. Сперва вернемся в код контроллера на который мы хотим вернуться и добавим ему специальный метод, который будет срабатывать при возвращении и который будет служить для нас помощником в переходе назад. Называется этот метод unwind. Если вы начнете набирать его в классе контроллера, вам появится подсказка с автозаполнением этого метода. Выбираем вариант без видимых аргументов (первый на моем скриншоте), после чего нам будет предложено дополнить имя этого метода, чтоб было понятно куда мы переходим. В качестве подсказки "name" я укажу "Root", имея ввиду корневой контроллер. Тело метода я оставлю пустым, так как



мне нужно просто вернуться назад, без какого-то дополнительного функционала. По факту это может быть любой метод, главное, чтобы это был именно @IBAction, принимающий в качестве аргумента сегвей (называться этот аргумент тоже может по всякому).

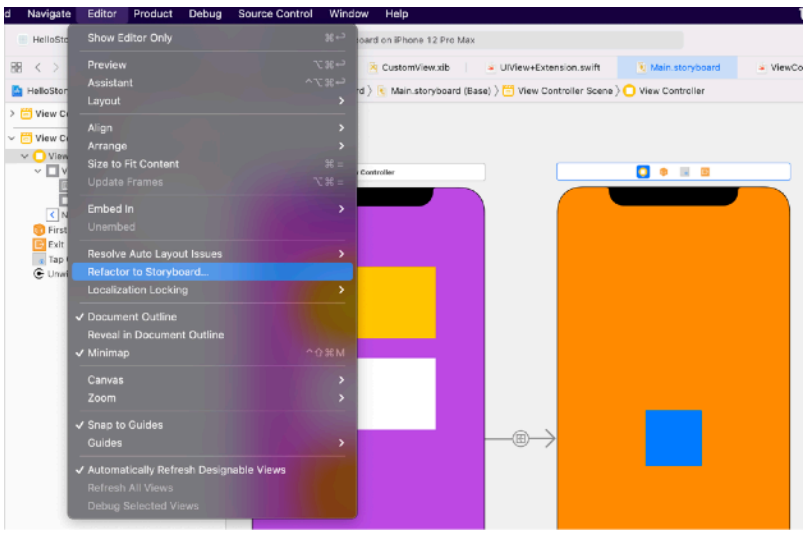
```
@IBAction func unwindToRoot(_ unwindSegue: UIStoryboardSegue) {}
```



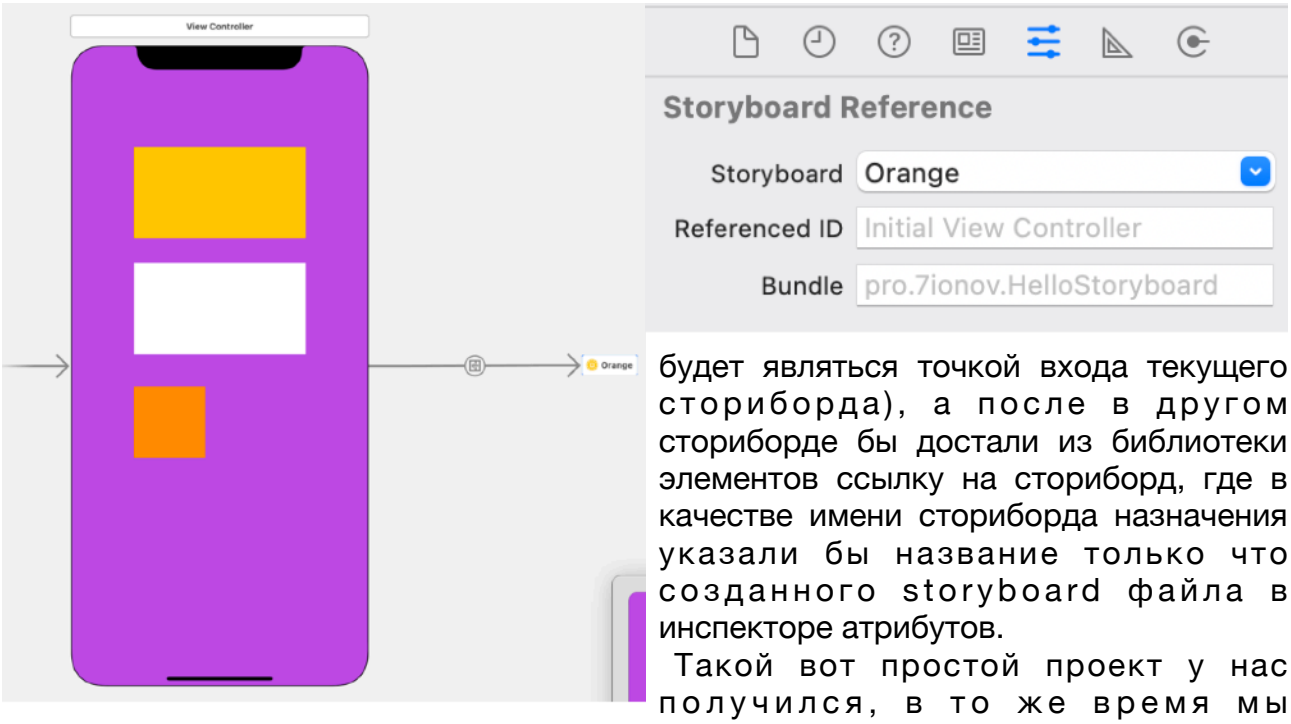
Осталось только вернуться в сториборд и вызвать данный метод по какому-либо действию (у нас это нажатие на синюю view оранжевого экрана). Поэтому с зажатым control перетаскиваем жест на иконку выхода (в иерархии или над экраном, крайняя правая), где будет предложено выбрать соответствующий метод, то есть куда мы хотим вернуться. Мы хотим вернуться на главный экран, поэтому выбираем здесь метод, описанный в классе контроллера главного экрана. Если бы у нас было несколько экранов, мы могли бы в каждом описать подобный метод и вернуться не обязательно на один шаг назад, а хоть и сквозь несколько экранов на самый первый.

Запустим приложение, поменяем цвета маленьких view на главном экране и попробуем перейти на оранжевый экран и назад. Теперь все работает как надо. У нас получилось сделать не только переход на другой экран, но и также вернуться назад.

Последнее небольшое улучшение для нашего проекта, которое мы рассмотрим это то, как разделить экраны на разные сториборды. Выше я писал, что крайне не желательно создавать на одном сториборде несколько экранов. Давайте же исправляться, вынесем оранжевый экран в отдельный storyboard файл. Это можно сделать как вручную, так и автоматизированно. Рассмотрим вариант автоматизированного выноса экрана в отдельный сториборд, тогда нам будет очевидно то, как бы мы делали это вручную. Выбираем контроллер оранжевого экрана в сториборде и в верхнем меню в разделе Editor выбираем пункт Refactor to



Storyboard. Нам будет предложено дать имя будущему сториборду, предлагаю назвать его просто Orange и нажимаем сохранить. Осталось только полученный файл расположить в той же группе (папке), что и остальные сториборды. Как видим у нас теперь есть отдельный сториборд со своей точкой входа, а если вернуться в Main.storyboard, то там вместо оранжевого экрана теперь появилась просто ссылка на другой сториборд (Storyboard Reference). Так что, если бы мы создавали новый сториборд вручную, мы бы сначала создали пустой файл с расширением .storyboard, добавили бы туда контроллер (не забыв указать, что он



будет являться точкой входа текущего сториборда), а после в другом сториборде бы достали из библиотеки элементов ссылку на сториборд, где в качестве имени сториборда назначения указали бы название только что созданного storyboard файла в инспекторе атрибутов.

Такой вот простой проект у нас получился, в то же время мы рассмотрели на нем базовые возможности разделения каждого экрана на контроллер и некоторое представление (view), то как мы можем использовать Interface Builder для визуального построения (верстки) экранов, а также то, как можно осуществлять навигацию между экранами приложения.

Примеры кода доступны по ссылке https://github.com/SemyonovE/Swift.Course.UIKit