Профессиональное программирование на Swift

Урок 9

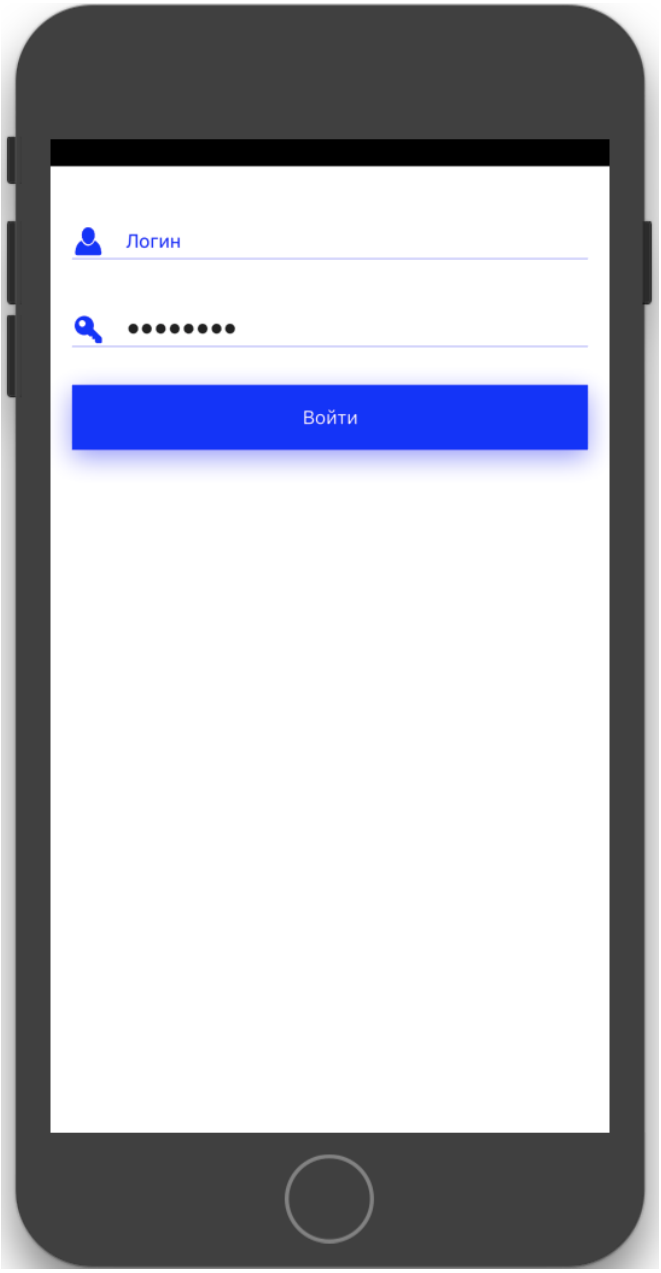
Композитный UI

Подходы по работе с UI. Storyboard или Xib?

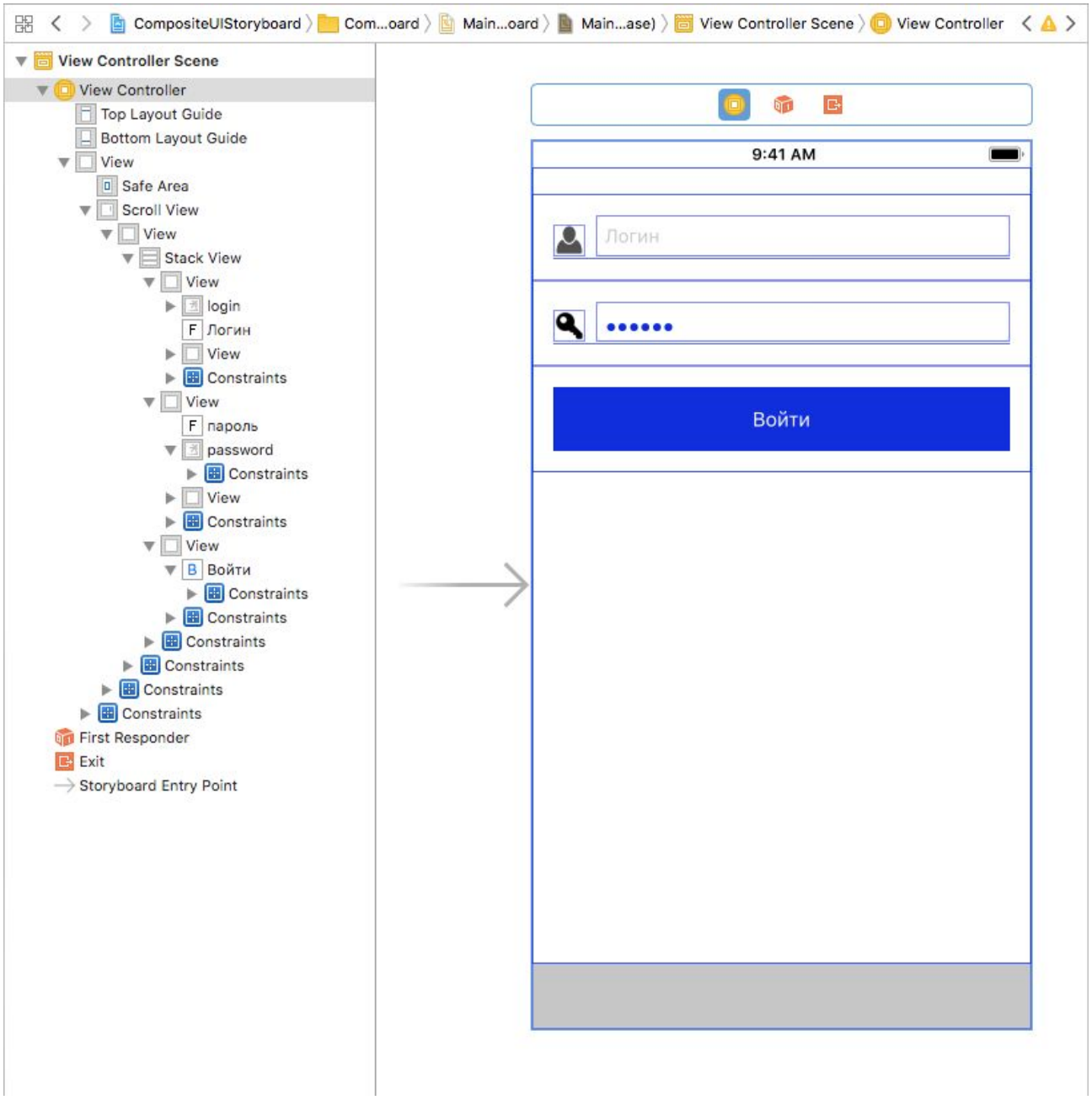
Как правильно собирать UI?

Что значит строить UI (пользовательский интерфейс) из кусочков? В предыдущих проектах мы использовали **storyboard** (в большинстве случаев — только один) и «накидывали» в него стандартные компоненты. Этот подход применим на маленьких проектах, но на больших не работает. Со временем появляются свои графические элементы и специфическая обработка данных внутри них. Эти элементы могут повторяться, причем не только на одной сцене **ViewController**, но и на других. И копировать их внешнее представление на **storyboard** не рекомендуется.

Например, при входе в приложение есть два поля ввода с ассоциированной иконкой, сепаратором (разделительной линией) снизу и кнопкой входа. Макет:



Казалось бы, простая задача. Первое, что приходит на ум — реализовать данную сцену в **storyboard** с использованием стандартных элементов:

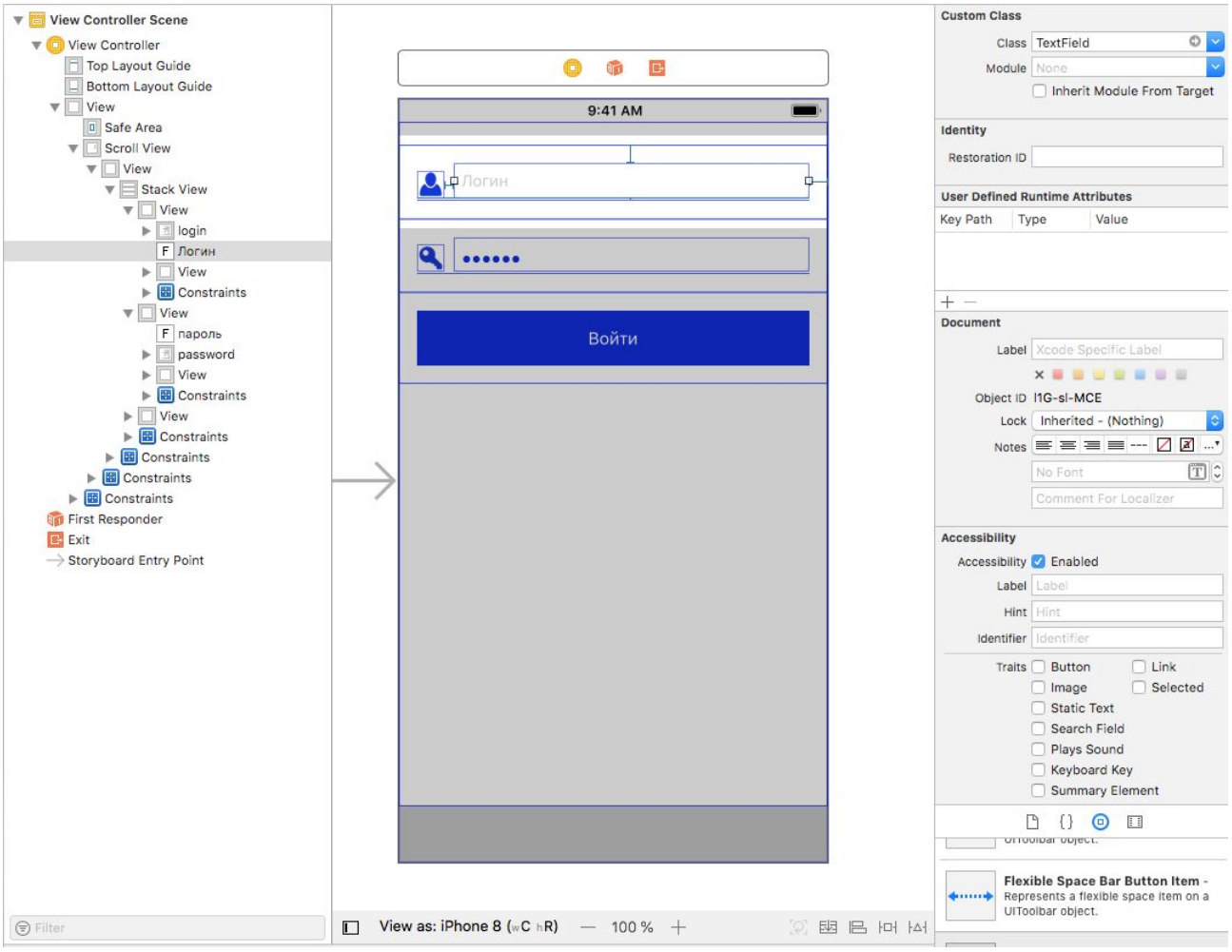


Далее создадим класс **TextField** наследника **UITextField**:

```
import UIKit

class TextField: UITextField {
    init(frame: CGRect, arg1: CGFloat, arg2: String) {
        super.init(frame: frame)

        configure()
    }

    required init(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)!
        configure()
    }

    private func configure() {
```

```
        borderStyle = .none
        adjustsFontSizeToFitWidth = true
        textColor = UIColor.black.withAlphaComponent(0.87)
        autocapitalizationType = .none
        autocorrectionType = .no
        spellCheckingType = .no

        let attributedHint = NSAttributedString(
            string: placeholder ?? "",
            attributes: [
                .font: UIFont.systemFont(ofSize: 14),
                .foregroundColor: UIColor.blue
            ])
        attributedPlaceholder = attributedHint
    }
}
```

И привяжем его к каждому **UITextField** на **storyboard**. Еще потребуем от дизайнера предоставить иконки для поля ввода логина и пароля в нужном цвете. Настроим вид для каждой разделительной линии. В результате получим нужный результат.



Но это решение сложно поддерживать и настраивать.

Минусы:

- Дублирование компонентов **View** для ввода данных;

- Сложно поддерживать единый стиль **View** для ввода данных;
- Вероятно повторное появление нового **View** для ввода данных на других сценах на **storyboard**.

Чтобы избежать подобной ситуации, придерживаемся следующих правил:

- Элементы интерфейса могут повторяться, делать каждый раз заново их не стоит;
- Выносим группу элементов в xib-файл;
- Для отдельных элементов стоит создавать свои классы (пример с классом **TextField**).

Когда использовать Xib-файлы?

Для ячеек таблицы. Все одинаковые ячейки, как и рассмотренный выше View ввода данных, подлежат выносу в класс и ассоциированный с ним xib-файл.

Например:

```
class SomeCell: UITableViewCell {
    @IBOutlet weak var nameLabel: UILabel! {
        didSet {
            nameLabel.font = UIFont.systemFont(ofSize: 14)
        }
    }

    @IBOutlet weak var someImageView: UIImageView! {
        didSet {
            someImageView?.image =
someImageView?.image?.withRenderingMode(.alwaysTemplate)
            someImageView.tintColor = UIColor.blue
        }
    }

    override func awakeFromNib() {
        super.awakeFromNib()
        self.selectionStyle = .none
    }
}
```

С последующим использованием в **UITableView**:

```
class SomeTableView: UITableViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        tableView.register(UINib(nibName: "SomeCell", bundle: nil),
forCellReuseIdentifier: "SomeCell")
    }

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell
    {
        if let cell = tableView.dequeueReusableCell(withIdentifier: "SomeCell")
```

```
as? SomeCell {
            cell.nameLabel.text = "Some"
            return cell
        }
        return UITableViewCell()
    }
}

class OtherTableView: UITableViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        tableView.register(UINib(nibName: "SomeCell", bundle: nil),
forCellReuseIdentifier: "SomeCell")
    }

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell
    {
        if let cell = tableView.dequeueReusableCell(withIdentifier: "SomeCell")
as? SomeCell {
            cell.nameLabel.text = "Other"
            return cell
        }
        return UITableViewCell()
    }
}
```

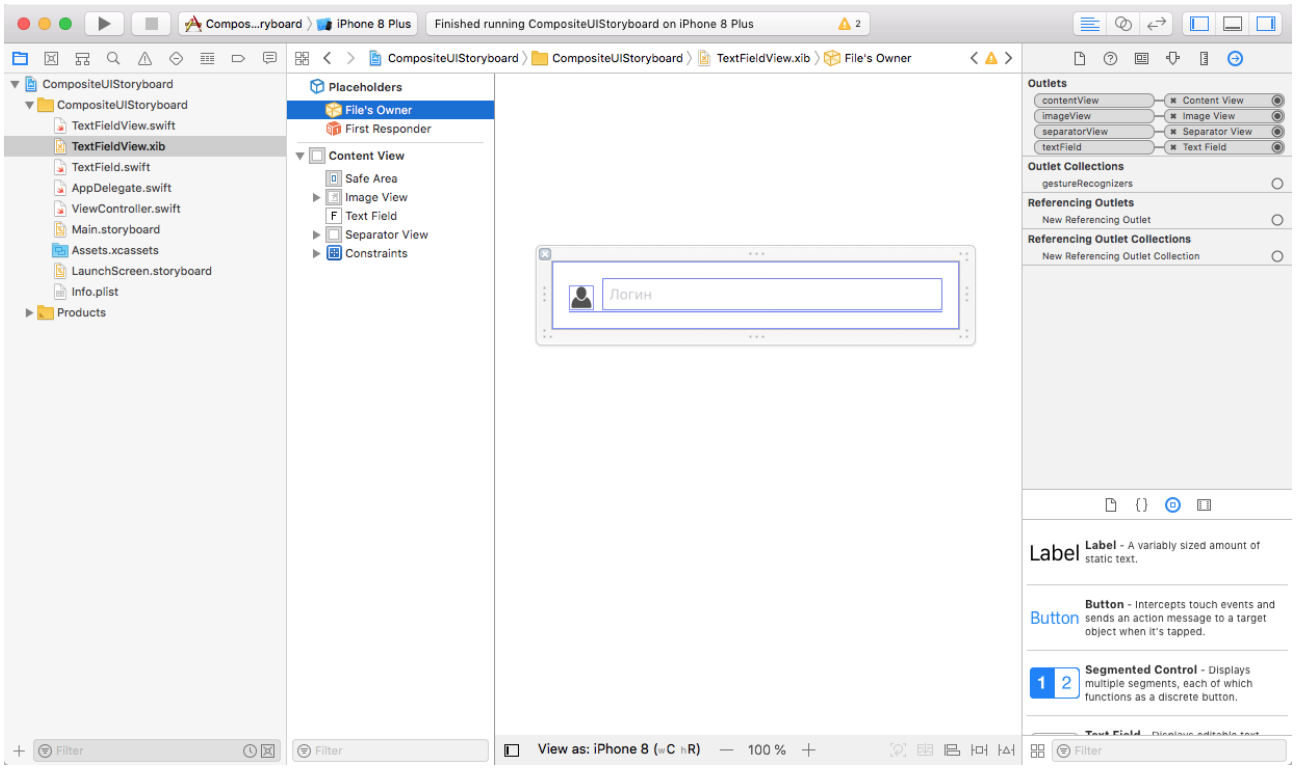
Для отдельных элементов стоит создавать свои классы, как в примере с классом **TextField**.

Доработаем пример входа в приложение с выносом **view** и вводом данных в отдельный xib-файл с ассоциированным классом.

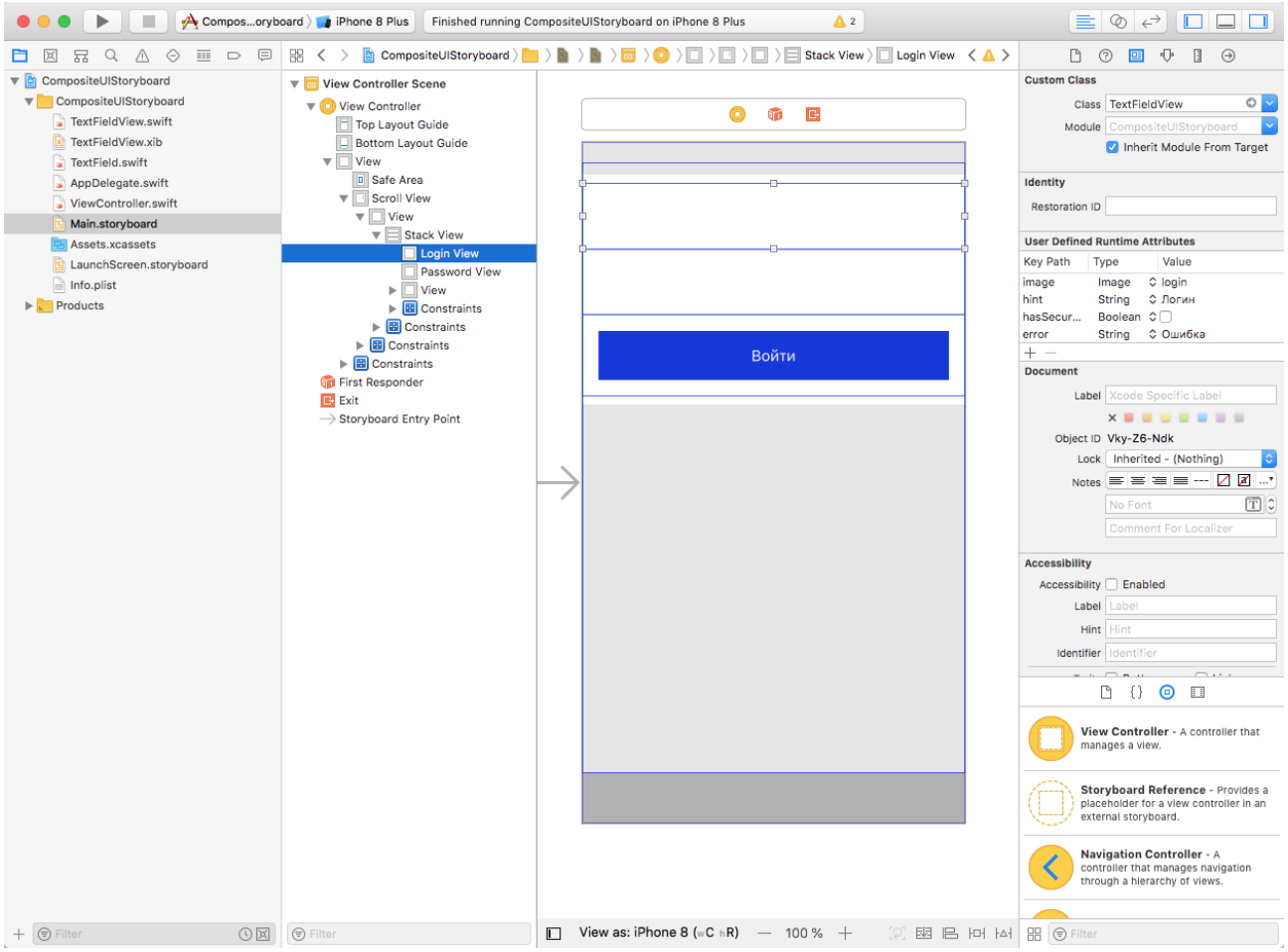
```
import UIKit

class TextFieldView: UIView {

    @IBOutlet weak var imageView: UIImageView! {
        didSet {
            if let image = imageView.image {
                imageView.image = image.withRenderingMode(.alwaysTemplate)
            }
            imageView.tintColor = UIColor.blue
        }
    }
    @IBOutlet weak var textField: TextField!
    @IBOutlet weak var separatorView: UIView! {
        didSet {
            separatorView.backgroundColor = UIColor.blue.withAlphaComponent(0.3)
        }
    }
    @IBOutlet weak var contentView: UIView!

    var image: UIImage? {
        get {
            return imageView.image
        }
```

```
        set {
            imageView.image = newValue?.withRenderingMode(.alwaysTemplate)
            imageView.isHidden = newValue == nil
        }
    }
    var hint: String {
        get {
            return textField.attributedPlaceholder?.string ?? ""
        }
        set {
            let attributedHint = NSAttributedString(
                string: newValue,
                attributes: [
                    .font: UIFont.systemFont(ofSize: 14),
                    .foregroundColor: UIColor.blue
                ])
            textField.attributedPlaceholder = attributedHint
        }
    }
    var text: String {
        get {
            return textField.text ?? ""
        }
        set {
            textField.text = newValue
        }
    }

    init(frame: CGRect, arg1: CGFloat, arg2: String) {
        super.init(frame: frame)

        configure()
    }

    required init(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)!
        configure()
    }

    private func configure() {
        Bundle.main.loadNibNamed("TextFieldView", owner: self, options: nil)
        addSubview(contentView)
        contentView.frame = self.bounds
        contentView.autoresizingMask = [.flexibleHeight, .flexibleWidth]
    }

}
```

Чтобы **view** из xib-файла подхватился на **storyboard**, нужно сделать связку с **IBOutlet contentView** и остальными **IBOutlets** через **Files Owner**:



Далее связываем класс **TextFieldView** с **storyboard**:



Получили отдельный класс с группой элементов для ввода данных, имеющих графическое отображение в xib-файле, для последующей легкой настройки.

Когда использовать Storyboard?

В **storyboard** очень удобно наблюдать общую картину приложения и взаимосвязи между его сценами. Можно отслеживать что угодно, потому что общий дизайн приложения содержится в одном файле, а не распределен между несколькими. **Мы легко настраиваем дизайн сцен.**

Storyboards могут описывают переходы между различными сценами с помощью **segues**, которые создаются соединением двух страниц в **storyboard**.

Благодаря segues пишем меньше кода для переходов между сценами пользовательского интерфейса. Правда не всегда с помощью **segues** можно полностью настроить сложную логику переходов, да и графически такое количество **segues** трудно для восприятия. Представьте, что для одного контроллера может быть до 10 различных переходов, и таких сцен множество на одном **storyboard**. Получится сложная и малопонятная навигация.

Storyboards облегчают работу с табличными типами, с ячейками. Можно создать таблицы практически полностью из **storyboard**, а это уменьшает код и время выполнения.

Рекомендация: используйте в проекте несколько **storyboards**, связанных одной функциональностью. У такого подхода следующие преимущества:

- Уменьшает время загрузки storyboard-файла в xcode;
- Облегчает совместную работу с реализацией разной функциональности;
- Уменьшает количество коллизий при использовании систем контроля версий.

Советы по реализации проектов:

Маленькие проекты — используйте **storyboard** без колебаний и реализуйте в ней все сцены. Время разработки сократится, и вы получите визуальное представление о всем приложении в одном месте. Это полезно с точки зрения организации.

Средние проекты — используйте **storyboard** для основной навигации. Применяйте xib-файлы для организации независимых **view**, которые будут использоваться повторно или нуждаются в конкретной конфигурации.

Большие проекты — необходимо разделить приложение на максимально возможное количество независимых модулей. Используйте несколько **storyboards**, по одному на каждый модуль, а также xib-файлы для организации независимых **view**, которые будут использоваться повторно или нуждаются в конкретной конфигурации. В подобных проектах рекомендуется программный подход реализации **view**.

Но в зависимости от проекта вам могут понадобиться все подходы реализации UI.

Когда использовать код?

Кодирование всех элементов пользовательского интерфейса дает ощущение контроля. Все, что можно сделать через **storyboard**, можно выполнить и с помощью кода. При этом вы точно знаете, какие настройки можете создать для элемента пользовательского интерфейса.

Всевозможные специфичные классы кнопок, текстовых меток, анимаций, которым не нужна видимая часть, можно реализовать программно. Например, кнопка с закруглением:

```
class SomeButton: UIButton {

    var cornerRadius: CGFloat = 0 {
        didSet {
            layer.cornerRadius = cornerRadius
            layer.masksToBounds = cornerRadius > 0
        }
    }

    var borderWidth: CGFloat = 0 {
        didSet {
            layer.borderWidth = borderWidth
        }
    }

    var borderColor: UIColor? = UIColor.clear {
        didSet {
            layer.borderColor = borderColor?.cgColor
        }
    }

}
```

Программный подход даст следующие преимущества:

- **Контроль**. Легко проводить настройку кода, так как мы точно знаем, где разместили все элементы пользовательского интерфейса;
- **Повторное использование.** Все созданные программно классы с легкостью можем применять по всему проекту;
- **Слияние конфликтов**. Использование систем контроля версий при этом подходе не вызывает сложностей, как при работе с **storyboard**.

Как это связать вместе?

Следуем правилу «Встраиваем меньшее в большее»: кнопку действия встраиваем в группу элементов, ее — в другую группу более высокого порядка, и в итоге встраиваемся непосредственно в контроллер.

Каждый **view** должен уметь конфигурировать себя: знать, какой шрифт использовать для своих компонентов, какие отступы, как отображать предоставленную информацию. Все это должно находиться в одном месте, чтобы не бегать по коду в поисках нужной команды.

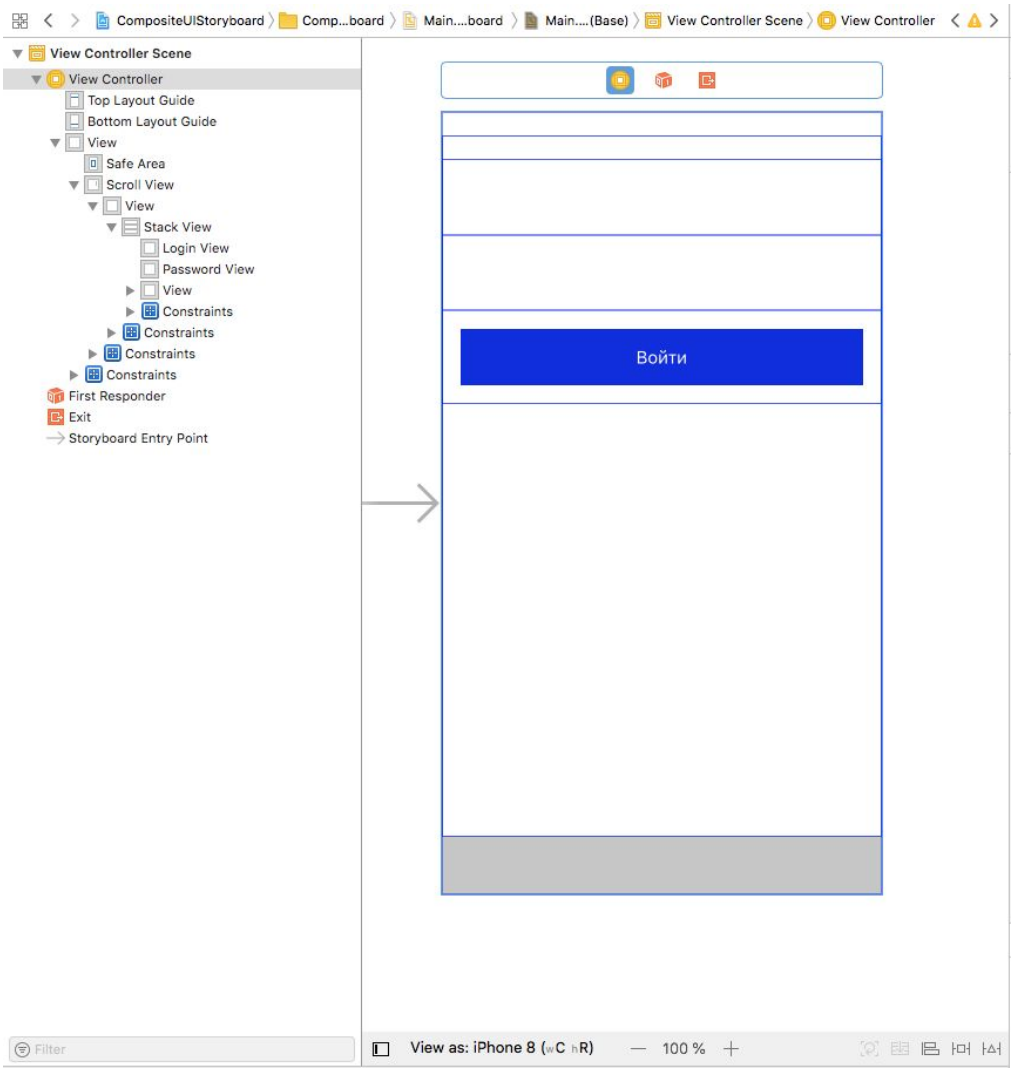
Хорошая практика использования фабрик — при создании графических компонентов, особенно если они создаются программно.

Например, фабрика создания разделительных линий может выглядеть следующим образом:

```
import SnapKit
import UIKit
```

```
protocol SeparatorFactoryAbstract: class {
    func makeTranslucent() -> UIView?
    func makeTranslucent(height: CGFloat) -> UIView?
    func makeDefaultGary() -> UIView?
    func makeDefaultGary(height: CGFloat) -> UIView?
}

final class SeparatorFactory: SeparatorFactoryAbstract {

    func makeTranslucent() -> UIView? {
        let view = UIView()
        view.backgroundColor = .clear
        return view
    }

    func makeTranslucent(height: CGFloat) -> UIView? {
        let view = makeTranslucent()
        configure(view: view, withHeight: height)
        return view
    }

    func makeDefaultGary() -> UIView? {
        let view = UIView()
        view.backgroundColor = UIColor(.cloudyBlue).withAlphaComponent(0.3)
        return view
    }

    func makeDefaultGary(height: CGFloat) -> UIView? {
        let view = makeDefaultGary()
        configure(view: view, withHeight: height)
        return view
    }

    func configure(view: UIView?, withHeight height: CGFloat) {
        guard let view = view else { return }

        view.snp.makeConstraints { maker in
            maker.height.equalTo(height)
        }
    }
}
```

Вернемся к примеру с формой входа. В последней модификации мы вынесли **view** ввода данных в отдельный **xib**, но при этом потеряли наглядность в **storyboard**.



На **storyboard** видим два пустых **view**. Только ассоциированные подписи дают представление об их назначении. Также нет возможности, внося изменения в параметры графических элементов, без запуска приложения смотреть результат этих изменений.

Применим программный подход и функциональность **IBInspectable** и **IBDesignable** (появились с XCode 6) и реализуем необходимую наглядность.

IBDesignable и **IBInspectable** — это способ создания пользовательских элементов и атрибутов с возможностью напрямую добавлять их в **Interface Builder** (xib-файл или **storyboard**).

Свойства, помеченные атрибутом **IBInspectable**, будут добавлены к атрибутам выполнения на вкладке **Attributes inspector**. То есть мы можем напрямую со **storyboard** или xib-файла задавать значения новых свойств.

Классы, помеченные атрибутом **IBDesignable**, непосредственно визуализируются на **storyboard** или xib-файле. Значения новых свойств, помеченные атрибутом **IBInspectable**, будут сразу применены и отображены.

Модифицируем код класса **TextFieldView** с использованием атрибутов **IBDesignable** и **IBInspectable**.

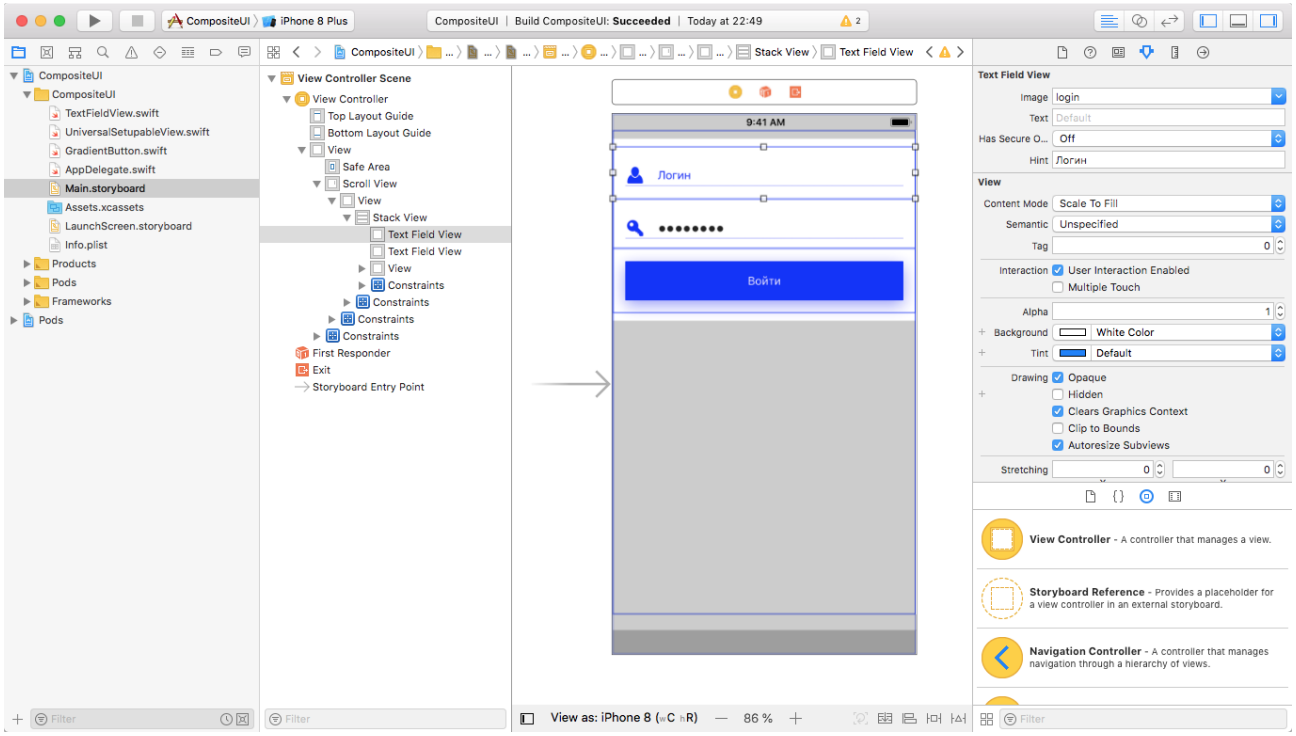
```
import SnapKit
import UIKit

@IBDesignable
```

```
class UniversalSetupableView: UIView {
    override init(frame: CGRect) {
        super.init(frame: frame)

        configure()
        setupConstraints()
    }

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

        configure()
        setupConstraints()
    }

    func configure() {
        assertionFailure("should override in children")
    }

    func setupConstraints() {
        assertionFailure("should override in children")
    }
}

final class TextFieldView: UniversalSetupableView {

    private let containerView: UIStackView = {
        let view = UIStackView()
        view.axis = .vertical
        return view
    }()

    private let dataContainerView: UIStackView = {
        let view = UIStackView()
        view.axis = .horizontal
        view.spacing = 16.0
        return view
    }()

    private let imageView: UIImageView = {
        let view = UIImageView()
        view.tintColor = UIColor.blue
        return view
    }()

    let textField: UITextField = {
        let view = UITextField()
        view.adjustsFontSizeToFitWidth = true
        view.textColor = UIColor.black.withAlphaComponent(0.87)
        view.autocapitalizationType = .none
        view.autocorrectionType = .no
        view.spellCheckingType = .no
        return view
    }()

    private lazy var separatorView: UIView = {
        let view = UIView()
        view.snp.makeConstraints { maker in
            maker.height.equalTo(1)
```

```
        }
        view.backgroundColor = UIColor.blue.withAlphaComponent(0.3)
        return view
    }()

    private lazy var springView: UIView = {
        let view = UIView()
        view.backgroundColor = UIColor.clear
        return view
    }()

    @IBInspectable
    var image: UIImage? {
        get {
            return imageView.image
        }
        set {
            imageView.image = newValue?.withRenderingMode(.alwaysTemplate)
            imageView.isHidden = newValue == nil
        }
    }

    @IBInspectable
    var text: String {
        get {
            return textField.text ?? ""
        }
        set {
            textField.text = newValue
        }
    }

    @IBInspectable
    var hasSecureOption: Bool = false {
        didSet {
            if hasSecureOption {
                textField.isSecureTextEntry = true
                textField.autocapitalizationType = .none
                textField.autocorrectionType = .no
                textField.clearButtonMode = .never
                textField.rightViewMode = .always
            }
        }
    }

    @IBInspectable
    var hint: String {
        get {
            return textField.attributedPlaceholder?.string ?? ""
        }
        set {
            let attributedHint = NSAttributedString(
                string: newValue,
                attributes: [
                    .font: UIFont.systemFont(ofSize: 14),
                    .foregroundColor: UIColor.blue
                ])
            textField.attributedPlaceholder = attributedHint
        }
```

```
    }

    override func configure() {
        dataContainerView.addArrangedSubview(imageView)
        dataContainerView.addArrangedSubview(textField)

        containerView.addArrangedSubview(dataContainerView)
        containerView.addArrangedSubview(separatorView)
        containerView.addArrangedSubview(springView)

        addSubview(containerView)
    }

    override func setupConstraints() {
        springView.snp.makeConstraints { maker in
            maker.height.equalTo(12).priority(999)
        }

        imageView.snp.makeConstraints { maker in
            maker.width.height.equalTo(24)
        }

        containerView.snp.makeConstraints { maker in
            maker.leading.equalToSuperview().offset(16)
            maker.top.equalToSuperview().offset(24)
            maker.trailing.equalToSuperview().offset(-16)
            maker.bottom.equalToSuperview()
        }
    }
}
```

В классе мы выделили несколько свойств (**image**, **text**, **hasSecureOption**, **hint**) с помощью атрибута **IBInspectable**. Мы их видим и можем изменить их значение на вкладке **Attributes inspector**. Также добавление атрибута **IBDesignable** в базовом классе **UniversalSetupableView** позволяет увидеть применение новых значений свойств, помеченных **IBInspectable**.



IBInspectable и **IBDesignable** — мощный инструмент визуализации графических компонентов. Используйте его в своих проектах.

Практическое задание

1. Реализовать экран просмотра товара с отзывами.

Дополнительные материалы

1. [Start Developing iOS Apps (Swift)](#).

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Магия IBDesignable или расширяем функциональность Interface Builder в Xcode](#).