По поводу протоколов и делегирования.

Постараюсь объяснить на примере реального проекта, а не сухой теории.

Итак, представьте, что у вас в проекте есть такая ситуация:

Есть некая кнопка, по нажатию на которую у вас идет запрос на удаленный сервер. Вы, выделили для таких запросов отдельный класс, назовем его CommunicationManager. Нажатие на кнопку вы обрабатываете во ViewController. Полученный ответ вам надо обработать и, допустим, поместить в какой-то UILabel на UI.

У вас получилось примерно так:

// ViewController

@IBAction func buttonGetDataFromRemoteServer(\_ sender: UIButton) {

let communicationManager: CommunicationManager = CommunicationManager();

let response = communicationManager.getSomeDataFromRemoteServer(stringParameter: "SomeString", intParameter: 42)

label.text = ("Hey, we've got the response: %@", response)

}

// CommunicationManager

func getSomeDataFromRemoteServer(stringParameter string: String, intParameter intp: Int) -> String {

// Запрос занимает время

Thread.sleep(forTimeInterval: 10);

return "Remote server response";

}

Тут есть одна проблема: запрос может выполняться долго (Thread.sleep это эмулирует), соответсвенно у вас повиснет UI на все время выполнения запроса. Решение - посылать такой запрос в отдельном потоке. Получится примерно так:

@IBAction func buttonGetDataFromRemoteServer(\_ sender: UIButton) {

let communicationManager: CommunicationManager = CommunicationManager();

let request\_queue = DispatchQueue(label: "RemoteServerRequest")

request\_queue.async {

communicationManager.getSomeDataFromRemoteServer(stringParameter: "SomeString", intParameter: 42)

}

}

func getSomeDataFromRemoteServer(stringParameter string: String, intParameter intp: Int) -> Void {

// do some stuff that takes time

Thread.sleep(forTimeInterval: 10);

DispatchQueue.main.async {

// Упс

}

}

"Упс", потому что мы теперь связались с асинхронностью и не можем так просто сделать return. ViewController у нас знает про CommunicationManager, но CommunicationManager не знает про ViewController. Как решать такую проблему?

Конкретно в Swift есть минимум четыре разных способа это сделать:

Способ №1 - **Делегирование**

Создаем новый протокол:

@objc protocol CommunicationProtocol {

@objc optional func parseResponse (response: String)

}

Наследуем от него наш ViewController и CommunicationManager

class ViewController: UIViewController, CommunicationProtocol {

// ...

@IBAction func buttonGetDataFromRemoteServer(\_ sender: UIButton) {

let communicationManager: CommunicationManager = CommunicationManager(withDelegate: self);

let request\_queue = DispatchQueue(label: "RemoteServerRequest")

request\_queue.async {

communicationManager.getSomeDataFromRemoteServer(stringParameter: "SomeString", intParameter: 42)

}

}

func parseResponse (response: String) {

label.text = ("Hey, we've got the response: %@", response)

}

class CommunicationManager: NSObject, CommunicationProtocol {

var delegate: CommunicationProtocol

init(withDelegate delegate: CommunicationProtocol) {

self.delegate = delegate

super.init()

}

func getSomeDataFromRemoteServer(stringParameter string: String, intParameter intp: Int) -> Void {

// do some stuff that takes time

Thread.sleep(forTimeInterval: 10);

DispatchQueue.main.async {

self.delegate.parseResponse!(response: "Remote server response")

}

}

}

Таким образом, у вас CommunicationManager *поручает* (собственно слово delegate и значит поручать) ViewController-у выполнить какой-то метод. При этом CommunicationManager по прежнему не имеет прямой ссылки на ViewController, т.е архитектуру мы не нарушаем.

Анатомия Constraint

Вся суть правил сводится к созданию вычисления у которого может быть только один ответ — расположение элемента интерфейса.  
  
Выглядит это приблизительно так:  
  
*Кнопка.Верх = ВысшаяТочкаИнтерфейса.Низ + 100*  
  
По данному выражению понятно что оно означает и какое правило устанавливает для *view*. Как уже было сказано, вычисления Auto Layout всегда происходят относительно ближайших элементов, будь это граница экрана или соседняя кнопка.  
  
В своих вычислениях *constraints* используют множители, ближайшие объекты и константы вроде + *100*из примера выше. Так же при создании правил не обязательно, чтобы это были равенства, вы можете использовать >= или<=.  
  
При создании *layout* желательно указывать два правила для каждого измерения на элемент. Но не забывайте про StackView, который вам очень поможет при создании интерфейса.  
  
Самым интересным фактом является то, что при создании *constraints* вы можете устанавливать приоритетность самих *constraints*. При вычислении, Auto Layout старается удовлетворить все *constraint*'ы в порядке приоритетности. Приоритет = 1000 — обязателен. Все остальные, менее приоритетные правила вы можете устанавливать для придания четкости обработке расположения элементов вашего интерфейса. В случае, если один из *constraint*'ов будет не правильно вычислен, Auto Layout использует ближайший *constraint* и начнет отталкиваться от него. Тем не менее, настоятельно рекомендую не перегружать интерфейс различными правилами и использовать дополнения только для достижения нужного результата.

Создание Auto Layout и его составляющих

Вы можете создавать *constraint*'ы 3-мя способами:  
  
1. CTRL + Перетаскивание, например, от *label* к верхней границе.  
2. Используя Stack, Align, Pin и Resolve Tools.  
3. Предоставить *Interface Builder* построить constraints вместо вас.  
  
Среди данных пунктов самый важный именно 2-й, так как использование этой панели является основным инструментом при создании разметки.  
  
**Stack** — собственно та самая кнопка, с помощью которой вы можете поместить выделенные детали интерфейса в StackView. Interface Builder сам решает каким будет StackView в зависимости от расположения элементов. Кроме кнопки Stack, StackView можно создать перетягиванием из библиотеки объектов, как любой другой элемент.  
  
**Align** — меню, которое позволит вам установить элементы четко по определенной линии, будь то с боку, вертикально по центру или снизу. Вы часто используете такой подход, когда выравниваете текст по центру или строго слева от начала страницы в текстовых редакторах.  
  
**Pin** — меню, позволяющее вам задать жесткие рамки относительно своего размера или ближайшего предмета. В нем вы выбираете какой *constraint* вы хотите задать в том или ином направлении и установить его параметры. Также с помощью данного меню можно, к примеру, придать группе кнопок одинаковый размер, не изменяющийся не смотря на масштаб экрана.  
  
**Resolve Tools** — самый лучший помощник в отладке *constraint*'ов. Основные возможности этого меню: убрать все правила, добавить предположительные *constraints*(Interface Builder построит все правила за вас), добавить отсутствующие *constraints*, обновить *constraints* или *frames*(положение объектов).  
Как вы видите, тут довольно много важных пунктов и они призваны облегчить все тяготы разработчика.  
  
Редактировать *constraint*'ы можно нажав на них в Interface Builder, найти в Size Inspector или в списке Document Outline. При редактировании параметров вы можете задавать идентификаторы для более легкого понимания и нахождения их в логах и консоли при выполнении различных отладок.  
  
Немаловажным аспектом при установке правил для элементов, являются параметры **CHCR** (Content-Hugging and Compression-Resistance Priorities) — эти параметры влияют на изменение самого элемента в зависимости от вышестоящего *view*. Грубо говоря *Hugging* — это нежелание элемента увеличиваться, а *Compression-Resistance* — нежелание уменьшаться. С помощью параметров CHCR можно к примеру изменять соотношение сжатия-расширения элементов в StackView в зависимости от размеров находящихся в нем элементов.  
  
Будьте внимательны — macOS и iOS рассчитывают *layout*'ы по разному: В macOS, Auto Layout может изменять размер окна и размер содержимого, а в iOS он может менять только размер содержимого, так как система сама определяет размер и границы приложения.

# **Основы использования Cocoapods в разработке приложений под iOS**

CocoaPods — это очень мощный инструмент, имеющий статус must have для любого iOS Developer'а. С помощью него мы можем легко, быстро и просто подключать различные вкусные тулзы, утилиты и библиотеки, которые значительно облегчают нам жизнь при разработке приложений. Но всяким инструментом надо уметь пользоваться и CocoaPods не исключение.  
Под катом информация, которая позволит начать освоение этого инструмента (основано на том, что изложено в документации и собственном опыте). И так, дорогой читатель, отложи в сторону микроскоп, хватит уже им гвозди заколачивать и продолжай читать. 

#### **Podfile**

**Podfile** — это как раз и есть та волшебная спецификация, в которой обозначены все зависимости необходимые для нашего проекта. Podfile всегда создает неявную цель (target), названную по умолчанию, которая связывает с первой целью (target'ом) проекта. **Этот файл должен быть назван просто Podfile и лежать в одной папке с файлом .xcodepoj.**  
  
**Для начала очень важно указать для какой платформы и какой ее версии ведется разработка.**Это делается вот таким образом:

platform :ios, '8.0'

В этом примере мы указываем, что разрабатываем приложение для iOS версии не ниже чем 8.0  
  
Далее мы можем скрыть все предупреждения от xCode, котоыре касаются подключаемых библиотек:

inhibit\_all\_warnings!

##### **Target проекта**

Если вы хотите использовать подключаемые библиотеки для нескольких target'ов (например для тестов), то для этого можно использовать команду **link\_with**:

platform :ios, '6.0'

link\_with 'MyApp', 'MyApp Tests'

pod 'AFNetworking', '~> 2.0'

pod 'Objection', '0.9'

В этом примере мы подключаем библиотеки не только для target'а нашего приложения, но и для тестов. Но **с этой командой нужно быть предельно осторожным**: эта команда подключает все библиотеки к двум target'ам, а иногда это может привести к ошибкам в работе, например не все привычные нам методы API будут работать для Application Extension (тех же виджетов — Today Extension).  
  
Более надежный способ — вручную задать target'ы для которых будут подключаться те или иные библиотеки:

def shared\_pods

pod 'AFNetworking'

pod 'INTULocationManager'

end

target 'myapp' do

shared\_pods

pod 'Reachability', '~> 3.1.1'

pod 'sqlite3', '~> 3.8.4.3'

end

target 'mywidget' do

shared\_pods

end

С помощью такого Podfile'а мы определили те библиотеки, которые будут у нас использоваться и для виджета и для приложения (shared\_pods). И обозначили какие именно надо подключать для какого target'а.

##### **Версия библиотеки**

Следующий момент тоже очень важен. Чаще всего мы кроме названия библиотеки дополнительно указываем номер версии, которую хотим использовать. Делаем это вот так:

pod 'WebViewJavascriptBridge', '~> 4.1.4'

Так мы это делаем как минимум потому, что подобные рекомендации нам как правило дают в ответе по запросу **pod search**. И с одной стороны это правильно, но сдругой стороны возможны нюансы.

Небольшой пример.  
  
Как избежать ненужных потерь нервных клеток? Иначе указать версию! Возможностей для этого много.  
  
Таким образом мы указываем точно какая версия нам нужна:

'WebViewJavascriptBridge', '4.1.4'

Кроме этого мы можем использовать и логические операторы, которые пояснять подпробно нет необходимости: '> 4.1.4' или '>= 4.1.4' или '< 4.1.4' или '<= 4.1.4'.  
  
Более подробно нужно рассказать об операторе '~>'. С помощью него мы можем указать версию, которая нам необходимо, но при этом получать нужные для нее bugfix'ы. Например вот так '~> 0.1.2' мы указываем, что хотим использовать версию библиотеки от 0.1.2 и до 0.2, **но не включая версию** 0.2. Или указав '~> 0.1' мы указываем, что планируем использовать все версии от 0.1 до 1.0, **но не включая версию** 1.0.  
Кроме этого мы можем указать ресурс репозитория с которого будем загружать библиотеку:

pod 'CRToast', :git => 'https://github.com/akhatmullin/CRToast.git'

Но и на этом еще не все. Указав ресурс мы еще можем указать нужную нам ветку,

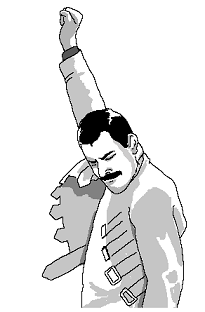
pod 'CRToast', :git => 'https://github.com/akhatmullin/CRToast.git', :branch => 'dev'

тег

pod 'CRToast', :git => 'https://github.com/akhatmullin/CRToast.git', :tag => '0.7.0'

или даже конкретный коммит

pod 'CRToast', :git => 'https://github.com/akhatmullin/CRToast.git', :commit => '082f8310af'



##### **Автоматизируй максимум или post\_install**

Иногда бывает так, что после установки той или иной библиотеки, нужно открыть тот или иной файл проекта, который был сгенерирован Cocoapods и добавить/убрать/изменить там какой-то параметр чтобы все работало корректно.  
  
Первое что нам приходит в голову в такой ситуации: сделать все как надо и написать в wiki на gitlab'е. Вроде все окей. А! Еще обязательно написать об этом всей команде в Skype… и сделать рассылку в почте! И ссылку на вики обязательно приложу…  
Сделано! А теперь все об этом забыли. Как быть? Использовать **post\_install**!

post\_install do |add\_app\_extension\_macro|

add\_app\_extension\_macro.project.targets.each do |target|

if target.name.include?("Pods-widget")

target.build\_configurations.each do |config|

config.build\_settings['GCC\_PREPROCESSOR\_DEFINITIONS'] ||= ['$(inherited)', 'AF\_APP\_EXTENSIONS=1']

end

end

end

end

В примере выше мы после установки всех необходимых библиотек перебираем в цикле все target'ы проекта автоматически создаваемого cocoapods, находим среди них нужный, а затем перебираем настройки этого target'а и найдя нужную добавляем нужную в build settings. В конкретном примере мы для AFNetworking устанавливаем параметр позволяющий использовать эту библиотеку в Application Extensions. И больше никаких записок в wiki. Совсем! Well done!

Alamofire

