Вступительная лабораторная работа

Привет!
Во-первых, мы очень рады, что вы успешно справились с тестом и решили попробовать выполнить вступительную лабораторную работу. Ее цель - выяснить как легко вы самостоятельно справитесь с хорошо описанным заданием.

Во-вторых, хотелось бы пояснить, что зачислены на курс будут студенты с наиболее качественной и полной лабораторной работой. Искренне надеемся, что, увлеченные выполнением этого задания, вы узнаете для себя много нового.

Задача: разработать приложение способное запросить и отобразить для выбранной пользователем валюты курс ее обмена на другую, так же выбранную пользователем валюту.

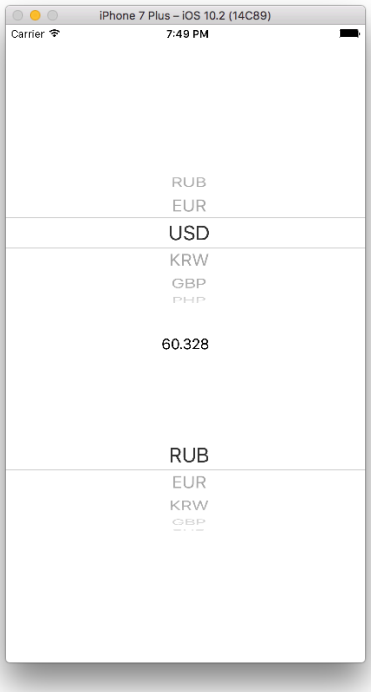
Проектирование интерфейса

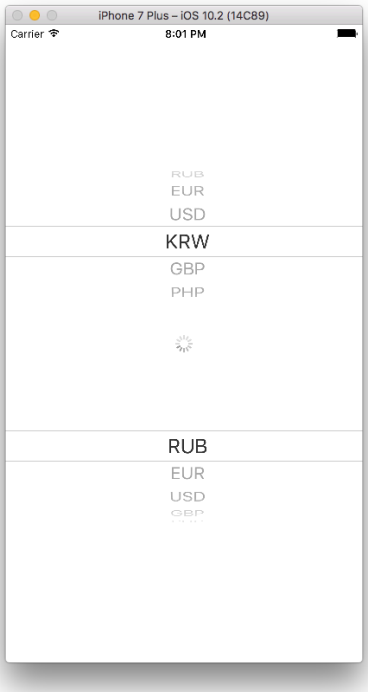
Для начала хочется определиться с тем, как будет выглядеть приложение и какой пользовательский кейс оно будет решать.

В нашем случае пользователю нужно предоставить возможность выбрать две валюты. Назовем их *“базовой”* и *“конечной”*. Единица *“базовой”* будет отображена в *“конечной”* валюте.

Например, если взять $ как “базовую” валюту, а ₽ как “конечную” то пользователь, после того как приложение запросит результат у сервера, должен увидеть приблизительно такое число: 60. Надеемся, оно будет даже меньше =).

Для того, чтобы сформировать визуальное представление забежим вперед и покажем как будет выглядеть экран нашего приложения с полученным курсом и во время выполнения запроса к серверу:





Разберем то, что отображено на экране несколько контролов (объекты, обычно имеющие визуальное представление в интерфейсе, позволяющие пользователю взаимодействовать с приложением):

1. Контрол для выбора валюты (x2)

Для этих целей выберем так называемый “пикер”. За его имплементацию отвечает класс UIPickerView. Его визуальное представление тебе скорее всего уже знакомо:

2. Контрол для отображения значения курса.

Это обычный “лейбл”. Его имплементация лежит в классе UILabel. Он может отображать текст в одну или несколько строк. В нашем случае он будет отображать курс валюты или текст ошибки выполнения запроса =).

3. Контрол для индикации активности приложения: выполнение сетевого запроса.

За его реализацию лежит в классе UIActivityIndicatorView. Выглядит так:

Отлично! Теперь мы понимаем из каких контролов будет состоять экран нашего приложения и как приблизительно он будет выглядеть. Перейдем к источнику данных, ведь нам нужны актуальные курсы валют.

Источник данных

Наше приложение будет отправлять http-запрос серверу и в ответ получать нужные данные (курс валюты).

Первый же запрос в Google по фразе “currency rate api” вернул нам линк на очень простой сервис, подходящий для нашей задачи http://fixer.io.

У этого сервиса есть метод (GET) http://api.fixer.io/latest?base=USD который возвращает JSON-объект с курсами валют по базовой (в данном примере по USD):

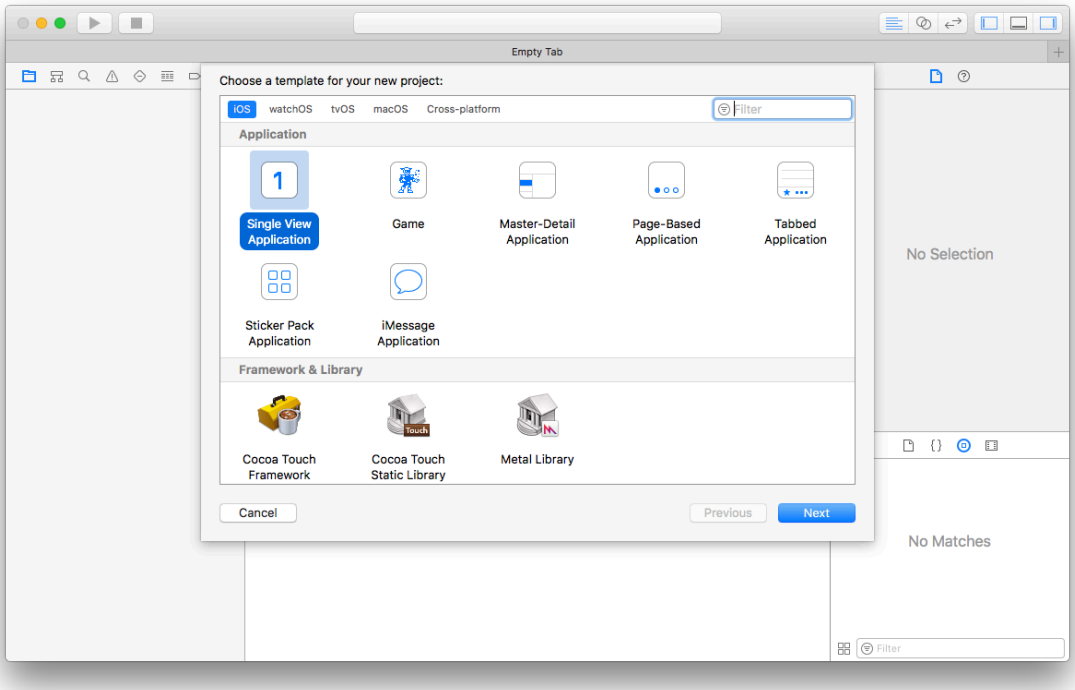
```
{
    "base": "USD",
    "date": "2017-01-11",
    "rates": {
        "AUD": 1.3559,
        "BGN": 1.8621,
        "BRL": 3.2168,
           …
        "RUB": 60.328,
           …
        "THB": 35.585,
        "TRY": 3.8796,
        "ZAR": 13.781,
        "EUR": 0.95211
    }
}
```

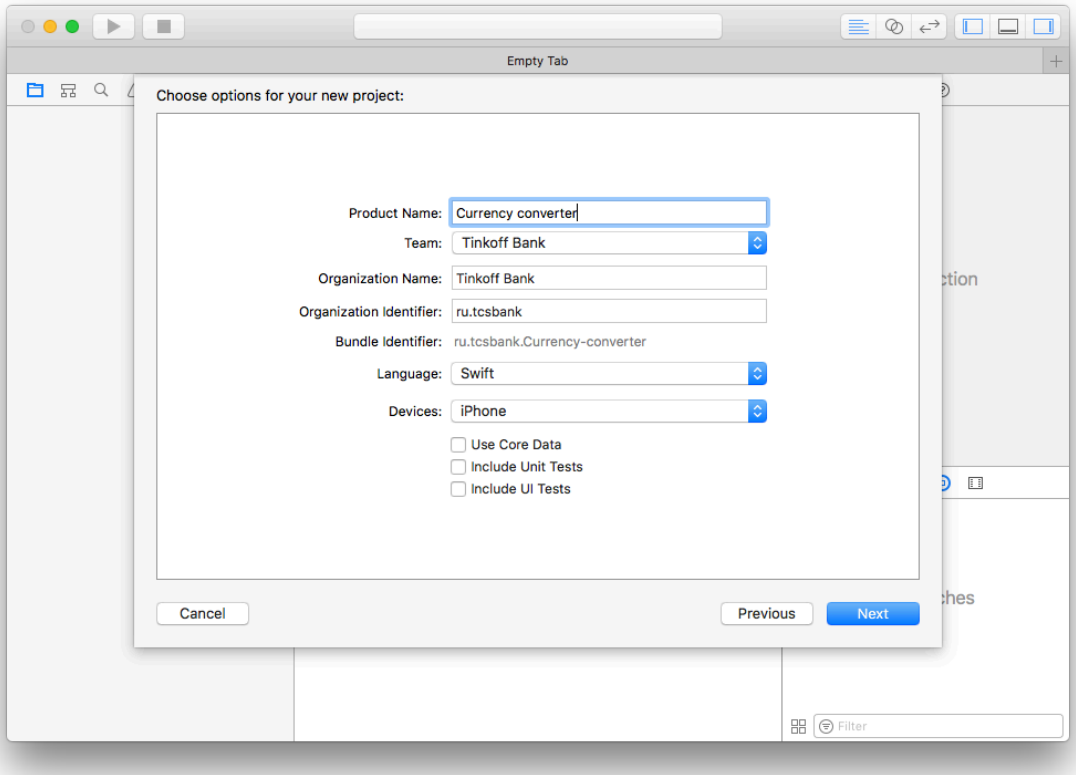
Из ответа-то мы и возьмем наше значение. С источником данных разобрались, теперь приступим к разработке. Начнем с создания проекта.

Создаем проект

Нам потребуется Xcode (при подготовке материала использовалась версия 8.2.1). Если он не установлен на вашем Mac (очень странно), то вам предстоит очень простой квест, который необходимо выполнить самостоятельно.

Итак, вы запустили Xcode и создаете новый проект. Выбираем шаблон iOS приложения с одной “вьюхой”:



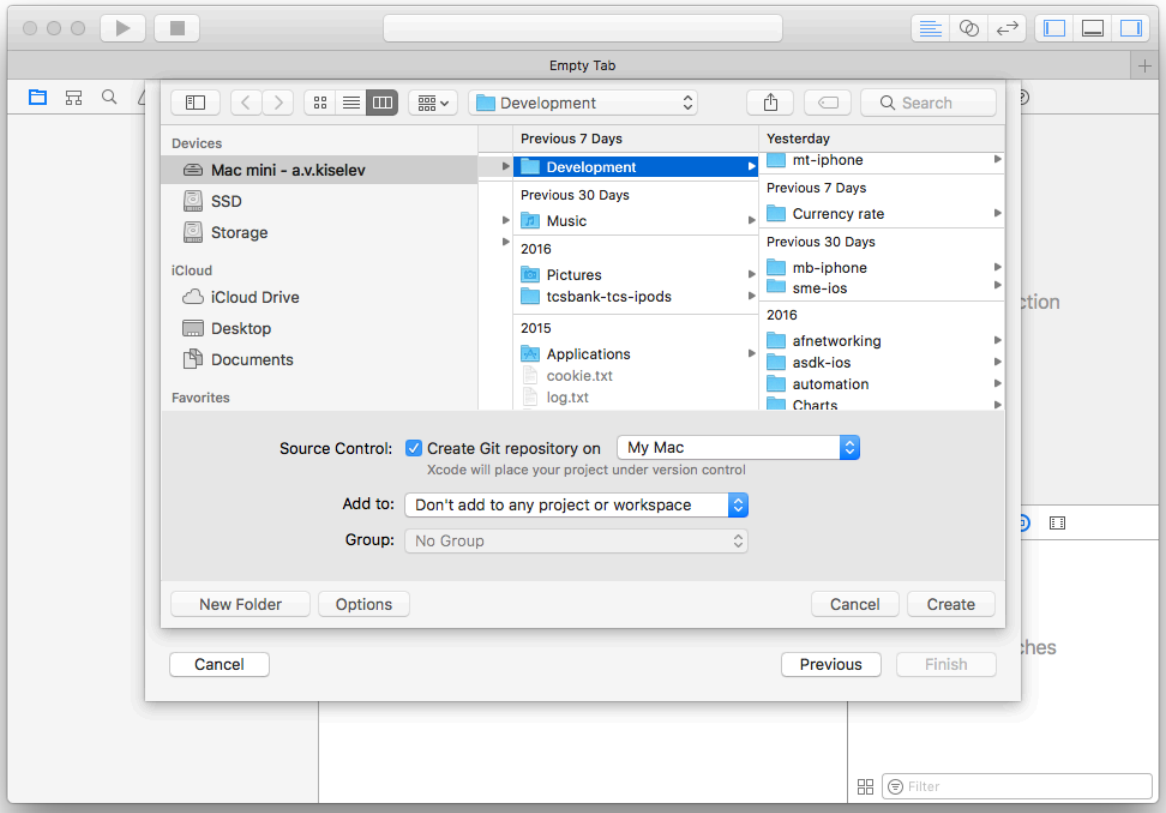


Далее указываем название продукта, команду разработки (если есть) и остальные поля (нужно погуглить их значение). Обязательно указываем язык Swift. Устройство оставляем iPhone.

Далее нужно выбрать где будет располагаться наш проект и стоит ли создавать для него git-репозиторий.

(Если вы умеете пользоваться git'ом - круто, тогда вопросов не возникнет. Если нет - будет неплохо если вы перейдете по ссылке и разберетесь с тем, что это такое и для чего это используется, а потом вернетесь к выполнению лабораторной работы.)

Выглядит так:



(Если вы уже имели дело с Xcode и знаете где тут "проектный навигатор" или "инспектор атрибутов" - это очень хорошо. Если нет - стоит погуглить видео или статью с обзором этой мощной среды разработки и затем вернуться к лабораторной работе. Обратите внимание на актуальную версию: **8.x.x** !)

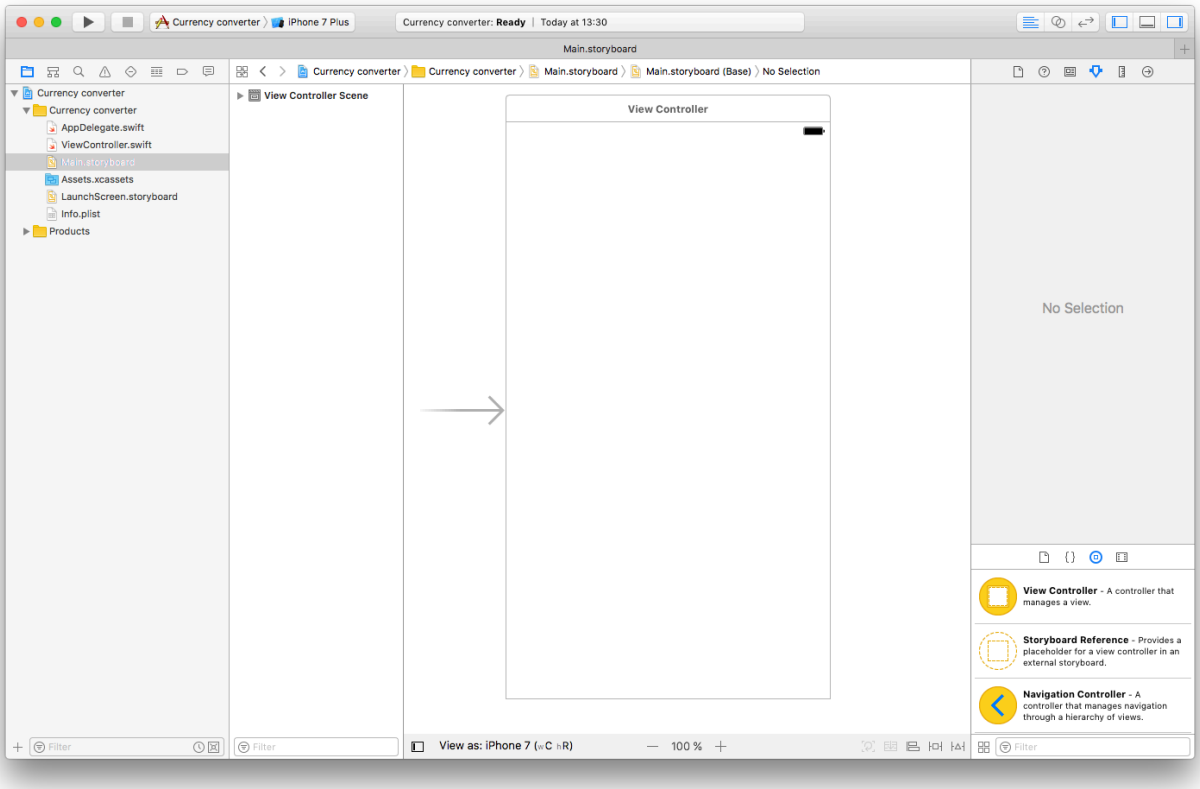
Отлично! Мы создали проект по шаблону и у нас уже есть несколько файлов в проектном навигаторе.(Если вы не знаете для чего нужны эти файлы - было бы неплохо с этим разобраться. Хотя бы поверхностно.)

Так как приложение крошечное и имеет очень простой интерфейс - давайте начнем с подготовки визуальных компонент.

Готовим интерфейс

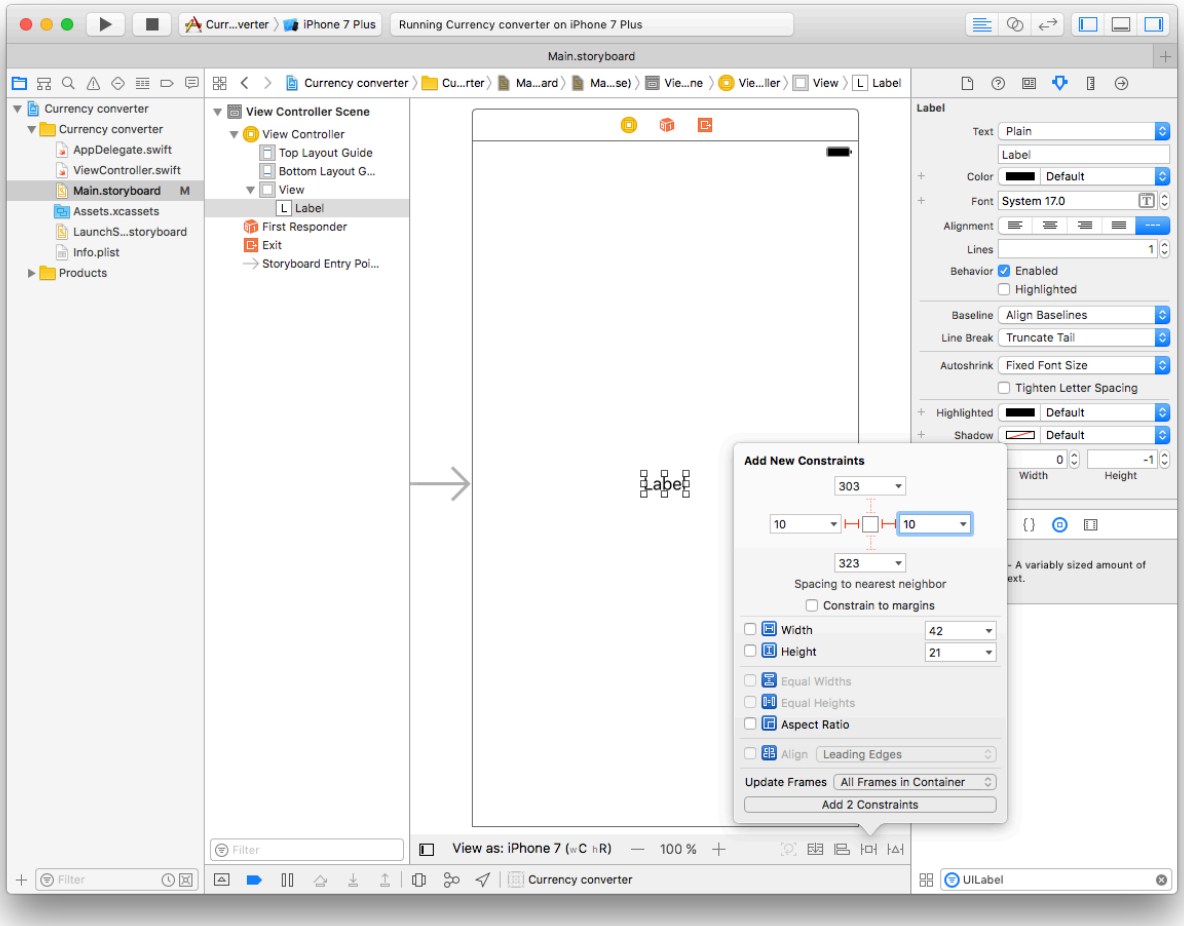
Ищем в проектном навигаторе файл *Main.storyboard* . Этот файл содержит в себе XML-разметку, которая описывает некоторые объекты приложения, их конфигурацию и взаимодействие. Используется “сториборд” чаще всего для удобного редактирования интерфейса приложения. В нашем случае в сториборде уже присутствует пустой *View Controller.*

View Controller - это объект, отвечающий за представление экрана нашего приложения. У него есть указатель на UIView который будет содержать в себе контролы, которые мы обозначили ранее.
Xcode автоматически откроет этот файл **.storyboard* визуальным редактором:

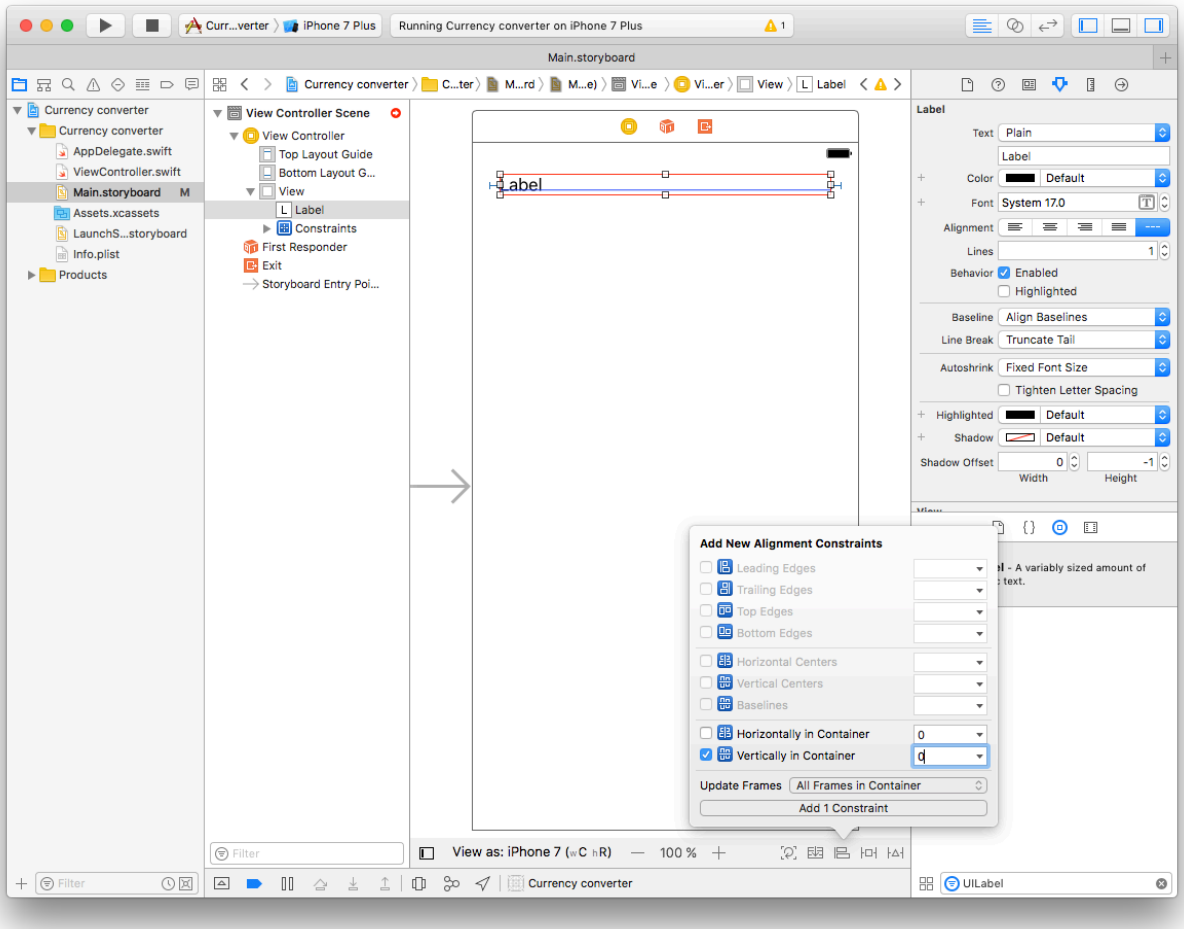


Нам нужно накидать компоненты на вьюху нашего View Controller’а. Начнем с текстового лейбла, который будет отображать нам курс валюты. Для этого:

1. Найдем компонент UILabel в библиотеке объектов (Object Library) и перетащим его в центр нашего ViewController’а.
2. При помощи механизма Auto Layout (Тут будет полезно погуглить, что это такое. Например, статья от Рея.) разместим его по центру вьюхи так, чтобы слева и справа от лейбла оставалось по 10 поинтов. Для этого:
   a. Выберем только что добавленный лейбл на экране.
   b. Добавим ему два “констрейнта” — зафиксировав отступ слева и справа от границ вьюхи. Для этого вызовем панель добавления констрейнт (см скриншот) и укажем величину отступов в ней (10). Обратите внимание на изображенные красным цветом иконки констрейнт — активны только сплошные (!). Так же выберем действие из выпадающего списка на панели, обновляющее фрейм контрола после добавления к нему констрейнт: “All Frames in container”.



c. После того как констрейнты были добавлены, а фрейм обновлен — можно увидеть, как лейбл расположился вверху экрана. Пофиксим это добавлением констрейнта выравнивания по вертикальной оси (не забудь обновить фреймы):

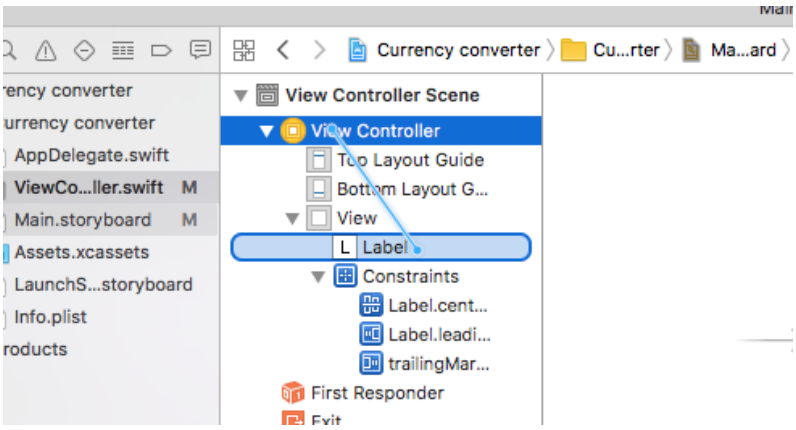


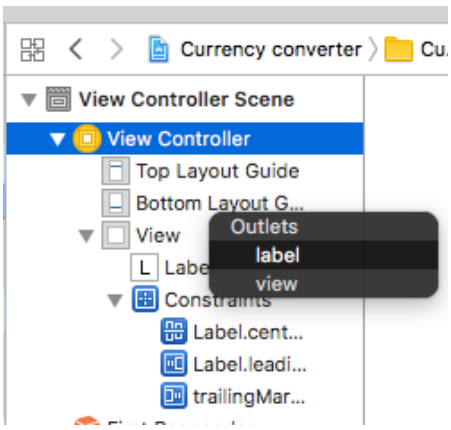
d. По умолчанию текст в лейбле “привязан” к левому краю. Мы же хотим сместить его к середине. Для этого в панеле инспектора атрибутов переключим Alignment на середину.

3. Указатель на добавленный лейбл нам нужно прокинуть в класс, отвечающий за реализацию ViewController’а, чтобы иметь возможность управлять его поведением “из кода”. Для этого нам нужно создать у ViewController’а специальное свойство с модификатором *@IBOutlet*, который позволит связать это свойство с конкретным объектом в сториборде. Итак:
   a. Выбираем файл *ViewController.swift* в файловом навигаторе и добавляем это свойство в наш класс. Назовем его “label”:

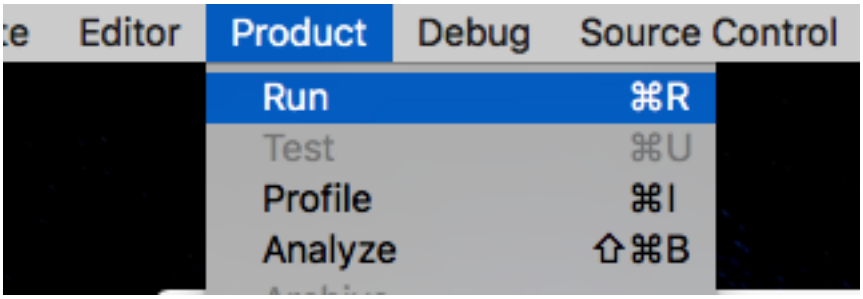
```swift
//
//  ViewController.swift
//  Currency converter
//
//  Created by a.v.kiselev on 12/01/2017.
//  Copyright © 2017 Tinkoff Bank. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }


}
```

   b. Возвращаемся к *Main.storyboard* и выбираем объект “View Controller” в списке объектов View Controller Scene.
   c. Зажав клавишу, **ctrl** тянем “связь” от объекта ViewController к объекту Label. Из доступных вариантов выбираем объявленный ранее “label“.

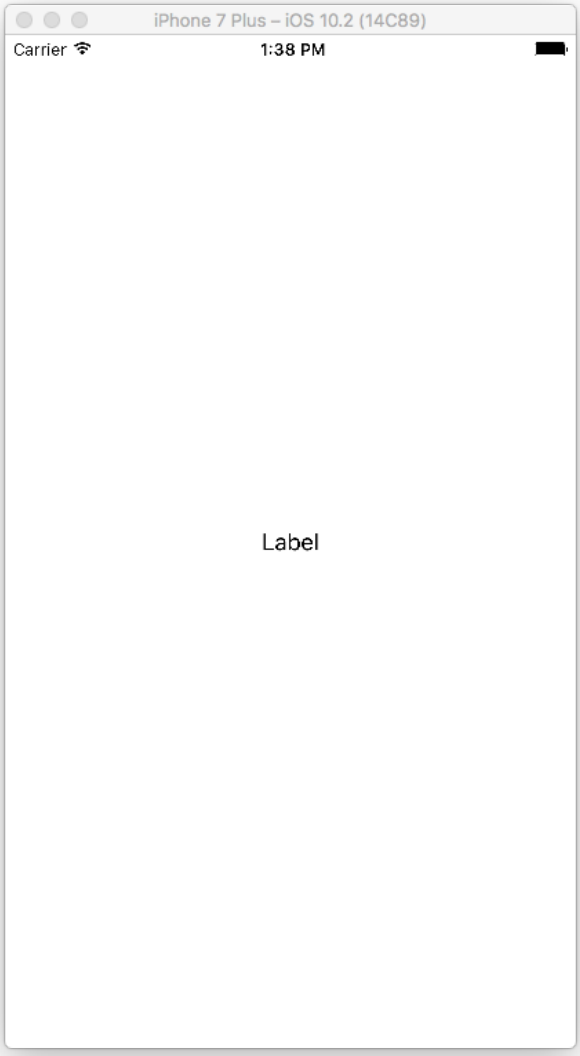




Теперь объект UILabel доступен внутри ViewController по указателю *label*. Попробуем собрать в симуляторе то, что получилось: жмите **Cmd+R** или выберите *Run* во вкладке *Product* основного меню Xcode.

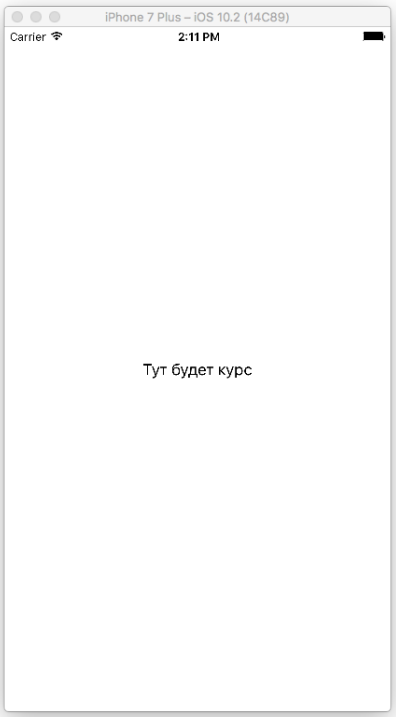


Через некоторое время должен запуститься симулятор с расположенным посередине экрана лейблом. Выглядеть это будет так (если не так - ищите ошибку):



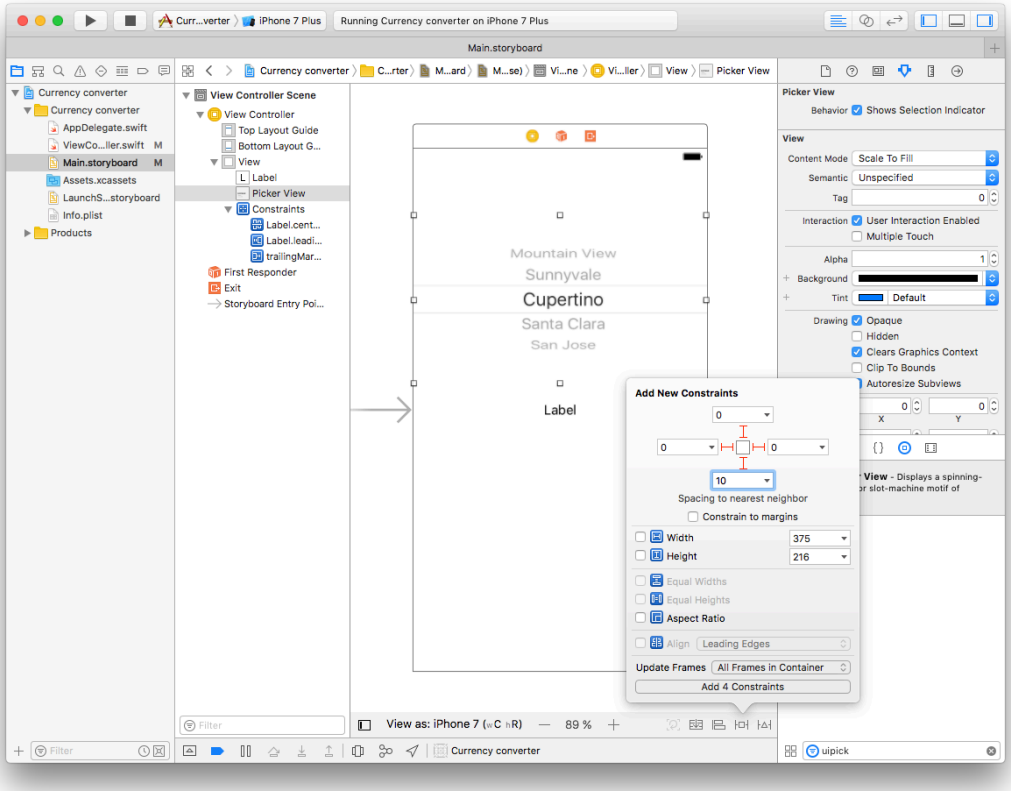
Но нам не хочется видеть текст “Label” на старте приложения, поэтому поменяем его. Для этого нужно обратиться к нашему лейблу внутри ViewController’а и задать ему текст: “Тут будет курс”. Выберем в проектном навигаторе файл *ViewController.swift* и в методе *viewDidLoad* (он вызывается когда экран будет загружен в память и все контролы на нем уже будут созданы) установим это значение свойству *text* нашему *label*.

```swift
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
        self.label.text = "Тут будет курс"
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }


}
```

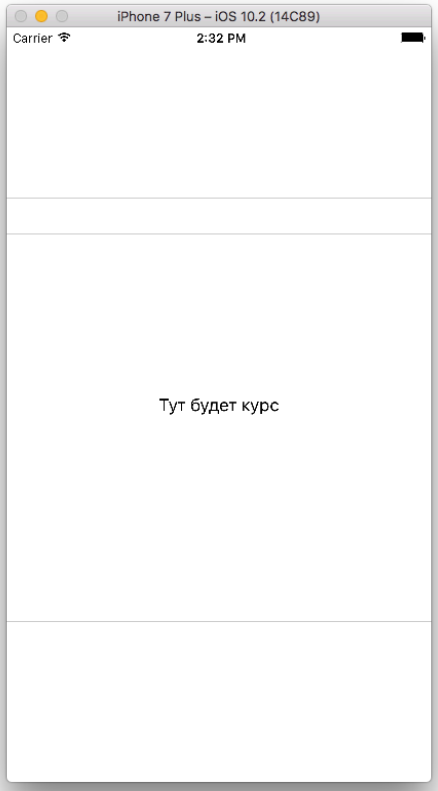


Итак, мы разобрались с тем, как “прокидывается” контрол в контроллер на примере нашего лейбла с курсом. Дальше нужно добавить “пикеры” для выбора валют и индикатор активности для того, чтобы пользователь видел, что приложение выполняет запрос. Начнем с пикеров:

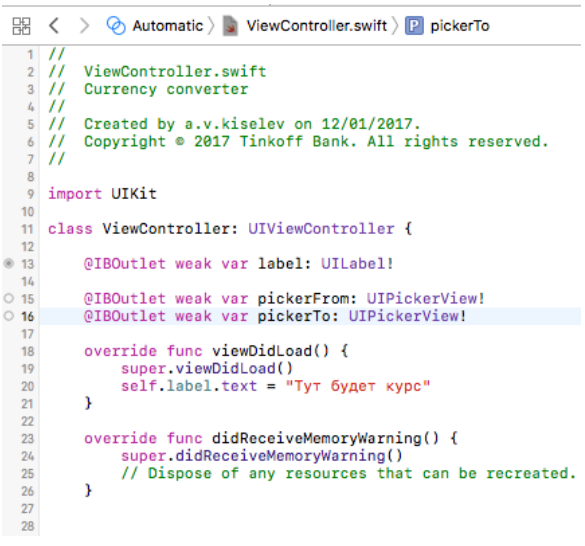
1. Найдем компонент UIPicker в библиотеке объектов (Object Library) и перетащим его в наш ViewController. Расположив его над лейблом.
2. При помощи механизма Auto Layout привяжем наш пикер к границам экрана сверху, слева и справа, а снизу - к лейблу на расстоянии 10 поинтов. Сделать это можно через панель добавления констрейнт (отображено на скриншоте).

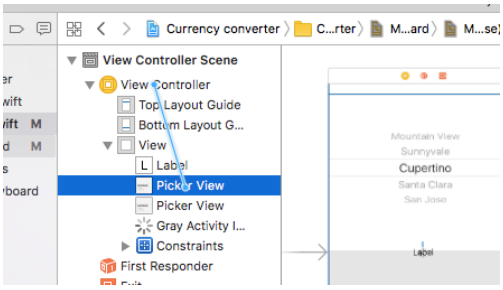


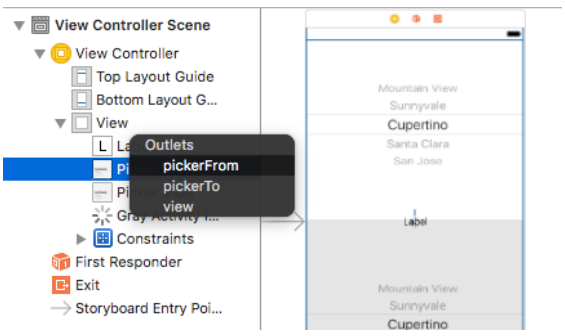
3. Аналогично добавим еще один пикер и прикрепим его “под” лейбл. После запуска должны получиться вот такие пустые пикеры как на скриншоте ниже.



4. Теперь нужно создать указатели (с модификатором @IBOutlet) на наши пикеры (для верхнего UIPickerView - pickerFrom, для нижнего - pickerTo) и установить для них связи:





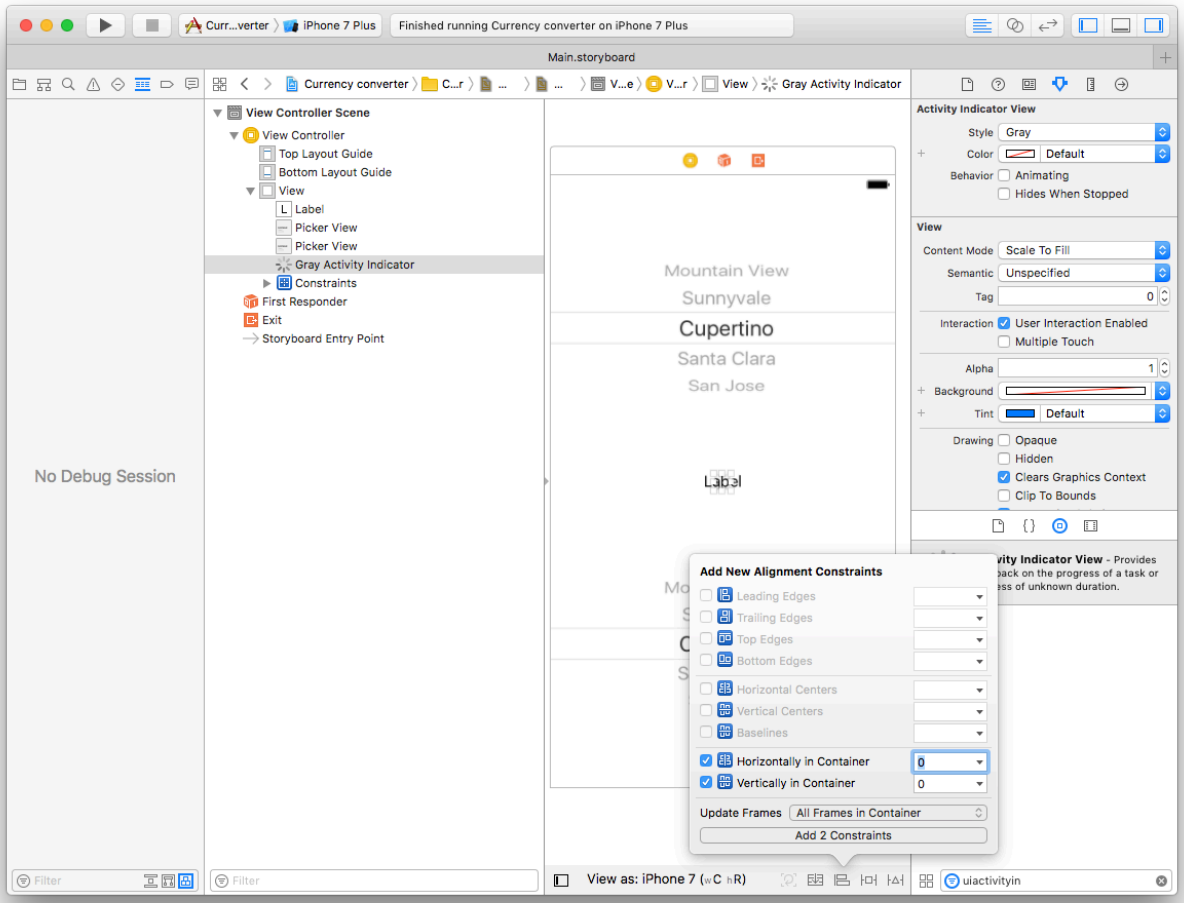


Осталось добавить индикатор загрузки:

1. Найдем компонент UIActivityIndicator в библиотеке объектов (Object Library) и перетащим его в наш ViewController. Расположим его посередине лейбла, т.к. он будет

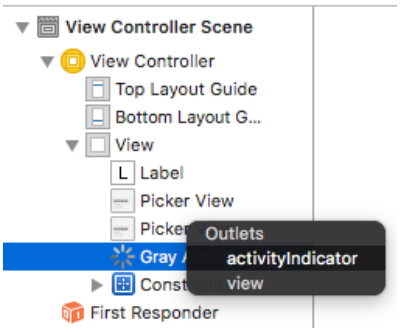
отображаться только тогда, когда у нас еще нет курса для выбранных валют, а значит лейбл пуст. И наоборот, когда курс есть - индикатор будем скрывать.

2. При помощи механизма Auto Layout привяжем наш индикатор активности к центру экрана, добавив ему констрейнты выравнивания в контейнере посередине вертикальной и горизонтальной осей:



3. Добавим указатель на UIActivityIndicatorView в наш ViewController и свяжем его с объектом в *Main.storyboard*.

```swift
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var label: UILabel!

    @IBOutlet weak var pickerFrom: UIPickerView!
    @IBOutlet weak var pickerTo: UIPickerView!

    @IBOutlet weak var activityIndicator: UIActivityIndicatorView!

    override func viewDidLoad() {
        super.viewDidLoad()
        self.label.text = "Тут будет курс"
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
```



Если вы все сделали правильно, то у вас есть все компоненты интерфейса и все они “прокинуты” в наш *ViewController*. Но все они пустые - данных нет. Давайте наполним наши пикеры валютой, чтобы у пользователя было из чего выбирать.

Наполняем пикеры валютой

Наши пикеры реализованы классом *UIPickerView*. Он спроектирован так, что для наполнения пикера данными необходимо указать объект, который бы подготовил данные для него. Такой объект называется *dataSource* и для корректной работы он должен реализовывать протокол UIPickerViewDataSource, который состоит из следующих методов:

1. *func numberOfComponents(in pickerView: UIPickerView) -> Int*
   Имплементация этого метода должна вернуть Int соответствующий числу компонент (вертикальных крутилок) нашего пикера. У нас она будет одна, так что тут мы вернем 1.
2. *func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int*
   Имплементация этого метода должна вернуть количество элементов в компоненте *component,* т.е. количество валют для выбора пользователю.

Ну что ж, нам нужно сохранить в массиве наименования валют, которые сможет выбирать пользователь. Начнем с простого массива *currencies* из нескольких валют:

```
18      @IBOutlet weak var activityIndicator: UIActivityIndicatorView!
19
20      let currencies = ["RUB", "USD", "EUR"]
21
22      override func viewDidLoad() {
23          super.viewDidLoad()
24          self.label.text = "Тут будет курс"
25
26      }
```

Теперь мы знаем сколько валют будет у пикера —сколько элементов в массиве *currencies*. А значит, мы можем добавить реализацию протокола *UIPickerViewDataSource* нашему ViewController'у. Для этого:

1. Объявим, что наш ViewController реализует этот протокол:

```
10
11 class ViewController: UIViewController, UIPickerViewDataSource {
12
```

2. Добавим реализацию этих методов:

```
36      //MARK: - UIPickerViewDataSource
37
38      func numberOfComponents(in pickerView: UIPickerView) -> Int {
39          return 1
40      }
41
42      func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
43          return currencies.count
44      }
```

3. Скажем пикерам кто тут у них dataSource:

```
22      override func viewDidLoad() {
23          super.viewDidLoad()
24          self.label.text = "Тут будет курс"
25
26          self.pickerTo.dataSource = self
27          self.pickerFrom.dataSource = self
28      }
```

Сколько валют-то мы знаем. А вот какие они - для этого нужен еще один объект, реализующий протокол UIPickerViewDelegate. От него нужно получить заголовок элемента. В нашем случае им снова станет ViewController. Итак:

1. Объявляем, что наш ViewController реализует UIPickerViewDelegate:

```
10
11 class ViewController: UIViewController, UIPickerViewDataSource, UIPickerViewDelegate {
12
```

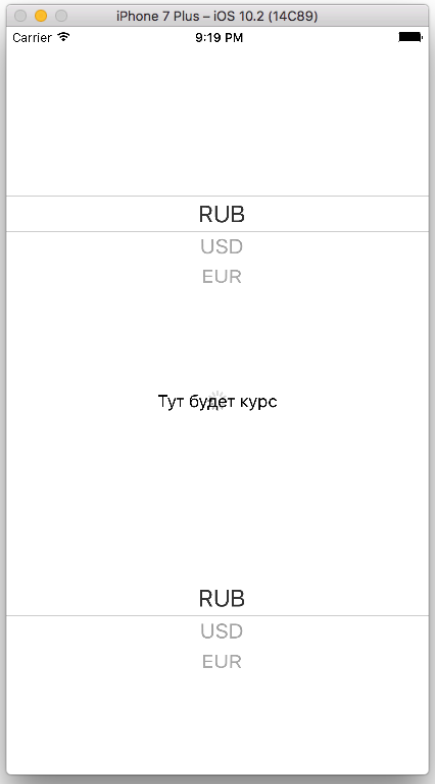
2. Добавляем реализацию метода, который будет возвращать строку, отображаемую пикером для строки с индексом row. В нашем случае это соответствующий индексу элемент массива *currencies:*

```
50      //MARK: - UIPickerViewDelegate
51
52      func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) ->
            String? {
53          return currencies[row]
54      }
```

3. Указываем нашим пикерам кто тут *delegate*:

```
22      override func viewDidLoad() {
23          super.viewDidLoad()
24          self.label.text = "Тут будет курс"
25
26          self.pickerTo.dataSource = self
27          self.pickerFrom.dataSource = self
28
29          self.pickerTo.delegate = self
30          self.pickerFrom.delegate = self
31      }
```

Пробуем запустить. Должно получиться так:



Если не получилось - ищите ошибки.

Не смотря на то, что пользователь теперь может потеребить пикеры — информацию о курсе он не получит. Нужно загружать данные из сети.

Получаем курс

Нам нужно написать метод, который бы мог отправить асинхронный http-запрос на сервер http://api.fixer.io/latest?base=USD (с выбранной пользователем базовой валютой), получить

ответ, а из ответа выбрать валюту в которую будет переведена базовая (из второго пикера).

Декомпозируем задачу начав с метода запроса валют по базовой, который бы получал в качестве входящего параметра кложур (блок) выполняющий парсинг (чтобы вынести логику парсинга ответа из метода).

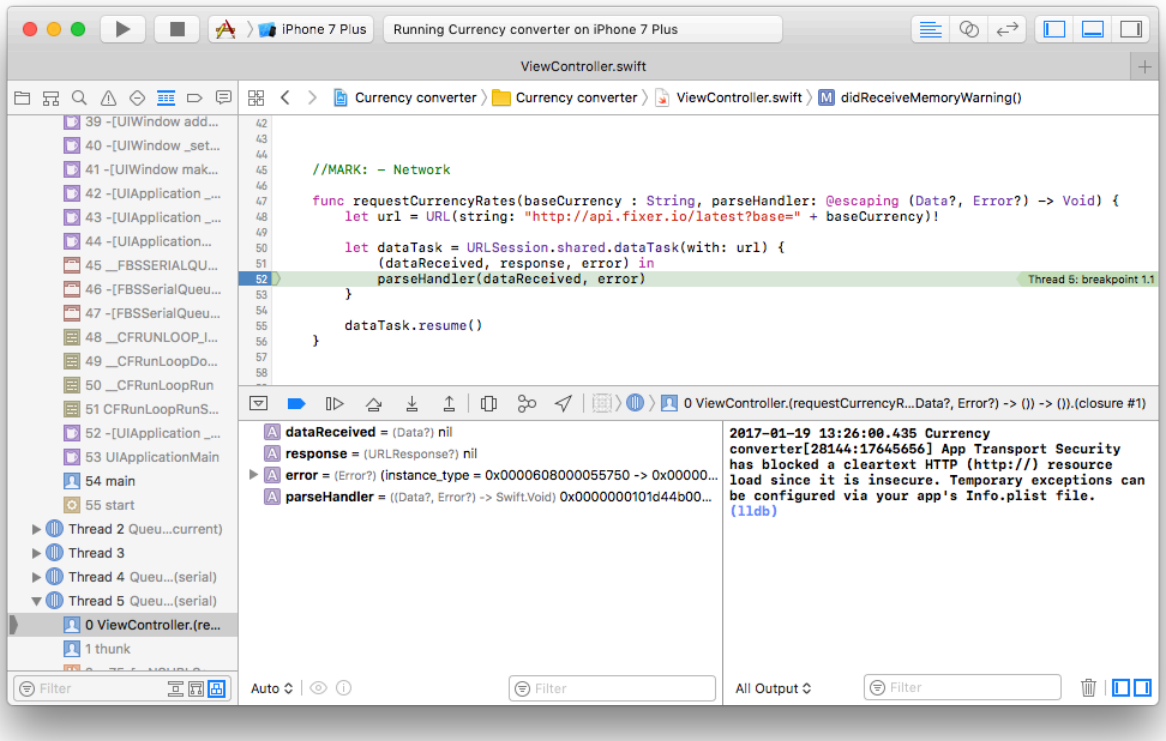
Воспользуемся объектом URLSession для загрузки данных: создадим с его помощью объект задачи загрузки данных URLSessionDataTask, сконфигурировав и передав нужный URL. Полученную задачу запустим на выполнение:

```
46
47      func requestCurrencyRates(baseCurrency : String, parseHandler: @escaping (Data?, Error?) -> Void) {
48          let url = URL(string: "http://api.fixer.io/latest?base=" + baseCurrency)!
49
50          let dataTask = URLSession.shared.dataTask(with: url) {
51              (dataReceived, response, error) in
52              parseHandler(dataReceived, error)
53          }
54
55          dataTask.resume()
56      }
```

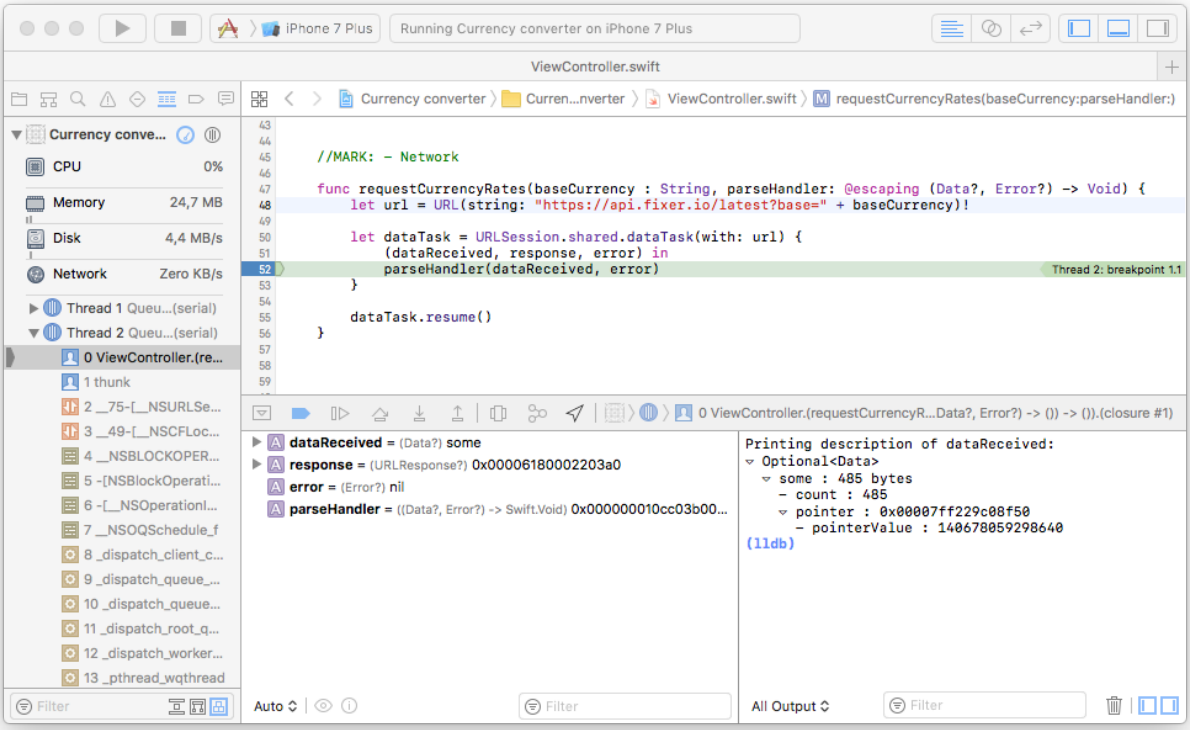
Проверяем, что запрос выполняется. Вызовем метод после загрузки ViewController'а:

```
22      override func viewDidLoad() {
23          super.viewDidLoad()
24          self.label.text = "Тут будет курс"
25
26          self.pickerTo.dataSource = self
27          self.pickerFrom.dataSource = self
28
29          self.pickerTo.delegate = self
30          self.pickerFrom.delegate = self
31
32          self.requestCurrencyRates(baseCurrency: "RUB") { (data, error) in
33
34          }
35      }
```

Самый простой способ посмотреть, как выполнился запрос - поставить breakpoint на строку в которой вызывается *parseHandler* внутри метода *requestCurrencyRates*:



В консоли видим, что App Transport Security заблокировал небезопасный запрос по http. Как и обещали в Apple, они действительно заблокировали к концу 2016 года все http-запросы из приложений, обязав использовать http**s**. Чтобы все-таки получить данные, нужно исправить запрос. Для этого достаточно просто отредактировать url запроса:



Теперь, когда мы получаем ответ и в *dataReceived* что-то есть - перейдем к реализации метода парсинга: попробуем из полученной Data получить json-объект и уже из него - значение курса для выбранной пользователем валюты. В случае, если ответ был невалидным - сформируем строку ошибки, которую будем отображать в лейбле.
(Для легкости восприятия посмотрите ответ сервера на этот запрос в главе “Источник данных”.)

Воспользуемся стандартным JSONSerialization и его методом *jsonObject(with: data!, options: [])* , который может выкинуть исключение, поэтому обернем его в do .. try .. catch конструкцию. В случае успешного получения json-объекта необходимо найти в словаре “rates” курс для выбранной пользователем валюты *toCurrency*:

```
63    func parseCurrencyRatesResponse(data: Data?, toCurrency: String) -> String {
64        var value : String = ""
65
66        do {
67            let json = try JSONSerialization.jsonObject(with: data!, options: []) as? Dictionary<String, Any>
68
69            if let parsedJSON = json {
70                print("\(parsedJSON)")
71                if let rates = parsedJSON["rates"] as? Dictionary<String, Double>{
72                    if let rate = rates[toCurrency] {
73                        value = "\(rate)"
74                    } else {
75                        value = "No rate for currency \"\(toCurrency)\" found"
76                    }
77                } else {
78                    value = "No \"rates\" field found"
79                }
80            } else {
81                value = "No JSON value parsed"
82            }
83        } catch {
84            value = error.localizedDescription
85        }
86
87        return value
88    }
```

Итак, у нас есть метод, который парсит Data из ответа запроса и возвращает строку, которую смело можно класть в наш лейбл.

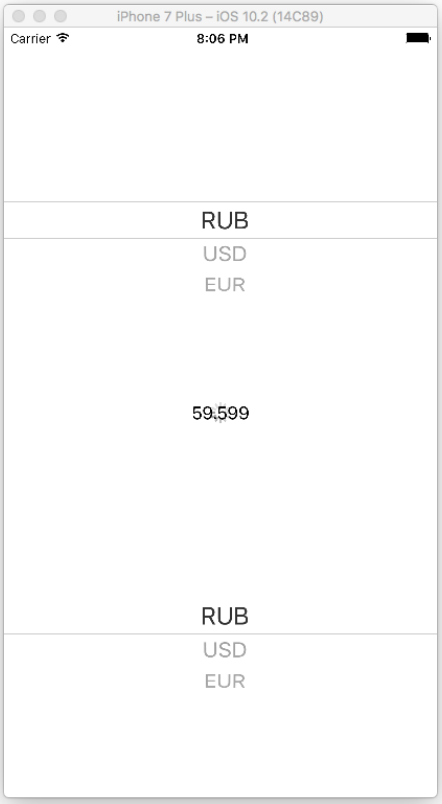
Теперь добавим парсинг в наш запрос и сделаем один общий удобный объединяющий метод:

```
97      func retrieveCurrencyRate(baseCurrency: String, toCurrency: String, completion: @escaping (String) -> Void) {
98          self.requestCurrencyRates(baseCurrency: baseCurrency) { [weak self] (data, error) in
99              var string = "No currency retrieved!"
100
101             if let currentError = error {
102                 string = currentError.localizedDescription
103             } else {
104                 if let strongSelf = self {
105                     string = strongSelf.parseCurrencyRatesResponse(data: data, toCurrency: toCurrency)
106                 }
107             }
108
109             completion(string)
110         }
111     }
```

Вызовем его после загрузки контроллера для проверки (обратите внимание, что блок будет выполнен в бэкграунд потоке, а мы хотим обратиться к элементу UI - лейблу, который должен быть изменен в основном потоке, поэтому мы воспользовались DispatchQueue и поменяли значение текста в лейбле уже в основной очереди):

```
32          self.retrieveCurrencyRate(baseCurrency: "USD", toCurrency: "RUB") {[weak self] (value) in
33              DispatchQueue.main.async(execute: {
34                  if let strongSelf = self {
35                      strongSelf.label.text = value
36                  }
37              })
38
39          }
40      }
```

Если все выполнили правильно, то симулятор покажет вам что-то подобное:



Один нюанс: у нас отображается индикатор загрузки и он статичен. Давайте настроим его так, чтобы он отображался только во время анимации.

```
22      override func viewDidLoad() {
23          super.viewDidLoad()
24
25          self.pickerTo.dataSource = self
26          self.pickerFrom.dataSource = self
27
28          self.pickerTo.delegate = self
29          self.pickerFrom.delegate = self
30
31          self.activityIndicator.hidesWhenStopped = true
32
33          self.retrieveCurrencyRate(baseCurrency: "USD", toCurrency: "RUB") {[weak self] (value) in
34              DispatchQueue.main.async(execute: {
35                  if let strongSelf = self {
36                      strongSelf.label.text = value
37                  }
38              })
39          }
40      }
```

Теперь застывшего индикатора не будет.

Как можно заметить, курс, отображающийся в лейбле не совпадает с выбранными в пикерах. Даже если выбрать другую валюту - курс не изменится. Нам нужно обрабатывать выбор пользователя и выполнять новый запрос, если валюта поменялась.

Приступим к обработке взаимодействия с пользователем.

Взаимодействие с пользователем

Протокол UIPickerViewDelegate так же позволяет реализовать метод *pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int),* который будет вызван тем пикером, чей индекс выбранного элемента поменялся.

Реализуем этот метод, **перенеся** выполнение запроса по выбранной валюте в него (уберем его из *viewDidLoad:* и подставим в качестве параметров значения из пикеров):

```
130     func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int) {
131         let baseCurrencyIndex = self.pickerFrom.selectedRow(inComponent: 0)
132         let toCurrencyIndex = self.pickerTo.selectedRow(inComponent: 0)
133
134         let baseCurrency = self.currencies[baseCurrencyIndex]
135         let toCurrency = self.currencies[toCurrencyIndex]
136
137         self.retrieveCurrencyRate(baseCurrency: baseCurrency, toCurrency: toCurrency) {[weak self]
                (value) in
138             DispatchQueue.main.async(execute: {
139                 if let strongSelf = self {
140                     strongSelf.label.text = value
141                 }
142             })
143         }
144     }
```

Теперь можно выбирать разные валюты и получать курс для них. Но, как можно заметить, интерфейс позволяет выбрать такую же валюту перевода, как и базовую. Это ни к чему, уберем эту возможность. Для этого напишем метод, возвращающий массив валют за исключением выбранной первым (pickerFrom) пикером:

```
114     func currenciesExceptBase() -> [String] {
115         var currenciesExceptBase = currencies
116         currenciesExceptBase.remove(at: pickerFrom.selectedRow(inComponent: 0))
117
118         return currenciesExceptBase
119     }
```

И поправим реализацию метода *pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int* протокола UIPickerViewDataSource так, чтобы он возвращал меньшее количество валют для пикера pickerTo:

```
130     func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
131         if pickerView === pickerTo {
132             return self.currenciesExceptBase().count
133         }
134 
135         return currencies.count
136     }
```

Так же поправим метод *pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?* протокола UIPickerViewDelegate, чтобы он не возвращал базовую валюту, т.к. она выбрана первым пикером:

```
143     func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? {
144         if pickerView === pickerTo {
145             return self.currenciesExceptBase()[row]
146         }
147 
148         return currencies[row]
149     }
```

Казалось бы, все должно работать, но пикер pickerTo не обновляет свой список просто так и базовая валюта не исчезает. Добавим обновление элементов pickerTo при изменении выбранного элемента в pickerFrom, а так же поправим *toCurrency* валюты внутри, используя *currenciesExceptBase*:

```
151     func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int) {
152         if pickerView === pickerFrom {
153             self.pickerTo.reloadAllComponents()
154         }
155 
156         let baseCurrencyIndex = self.pickerFrom.selectedRow(inComponent: 0)
157         let toCurrencyIndex = self.pickerTo.selectedRow(inComponent: 0)
158 
159         let baseCurrency = self.currencies[baseCurrencyIndex]
160         let toCurrency = self.currenciesExceptBase()[toCurrencyIndex]
161 
162         self.retrieveCurrencyRate(baseCurrency: baseCurrency, toCurrency: toCurrency) {[weak self] (value) in
163             DispatchQueue.main.async(execute: {
164                 if let strongSelf = self {
165                     strongSelf.label.text = value
166                 }
167             })
168         }
169     }
```

Если переключать пикер - валюта загружается, но хочется, чтобы на старте приложения по выбранным валютам сразу выполнялся запрос. Для этого напишем метод выполнения обновления, в который вынесем из делегатного метода логику выполнения запроса, заменив ее вызовом этого метода:

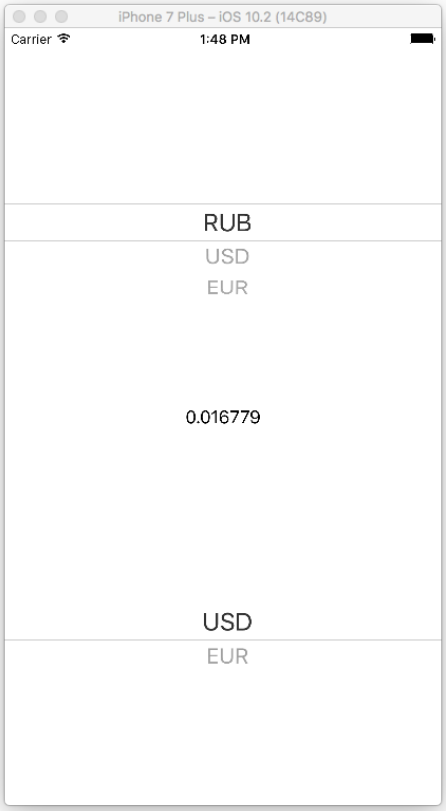
```
120     func requestCurrentCurrencyRate() {
121         let baseCurrencyIndex = self.pickerFrom.selectedRow(inComponent: 0)
122         let toCurrencyIndex = self.pickerTo.selectedRow(inComponent: 0)
123 
124         let baseCurrency = self.currencies[baseCurrencyIndex]
125         let toCurrency = self.currenciesExceptBase()[toCurrencyIndex]
126 
127         self.retrieveCurrencyRate(baseCurrency: baseCurrency, toCurrency: toCurrency) {[weak self] (value) in
128             DispatchQueue.main.async(execute: {
129                 if let strongSelf = self {
130                     strongSelf.label.text = value
131                 }
132             })
133         }
134     }
```

```
165     func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int) {
166         if pickerView === pickerFrom {
167             self.pickerTo.reloadAllComponents()
168         }
169 
170         self.requestCurrentCurrencyRate()
171     }
```

Вызовем его так же и после загрузки контроллера:

```
22      override func viewDidLoad() {
23          super.viewDidLoad()
24
25          self.pickerTo.dataSource = self
26          self.pickerFrom.dataSource = self
27
28          self.pickerTo.delegate = self
29          self.pickerFrom.delegate = self
30
31          self.activityIndicator.hidesWhenStopped = true
32          self.requestCurrentCurrencyRate()
33      }
```

Теперь после старта приложения мы сразу видим курс:



Включим наш индикатор активности и отчистим лейбл в момент начала выполнения запроса. По завершению запроса индикатор активности нужно выключить:

```
121     func requestCurrentCurrencyRate() {
122         self.activityIndicator.startAnimating()
123         self.label.text = ""
124
125         let baseCurrencyIndex = self.pickerFrom.selectedRow(inComponent: 0)
126         let toCurrencyIndex = self.pickerTo.selectedRow(inComponent: 0)
127
128         let baseCurrency = self.currencies[baseCurrencyIndex]
129         let toCurrency = self.currenciesExceptBase()[toCurrencyIndex]
130
131         self.retrieveCurrencyRate(baseCurrency: baseCurrency, toCurrency: toCurrency) {[weak self] (value) in
132             DispatchQueue.main.async(execute: {
133                 if let strongSelf = self {
134                     strongSelf.label.text = value
135                     strongSelf.activityIndicator.stopAnimating()
136                 }
137             })
138         }
139     }
```

В текущем состоянии приложение работает и им можно пользоваться - оно решает основную задачу. Но, безусловно, что-то можно было бы улучшить.

Что можно было бы улучшить

Вышеописанная лабораторная работа является лишь примером. Мы искренне надеемся, что у вас возникнет желание доработать приложение, продемонстрировав, на что вы способны.

Итак, с чего можно начать:

1. Можно заметить, что в момент, когда лейбл очищается и на экране виден индикатор активности — пикеры немного “прыгают” навстречу друг другу. Было бы здорово избавиться от этого “бага”.
2. На данный момент доступные валюты “захардкожены” в приложении. Было бы намного удобнее, если бы приложение формировало доступные валюты по запросу https://api.fixer.io/latest. Во-первых, было бы больше валют, а во-вторых — они бы обновлялись без обновления приложения. (Не забудьте обработать кейс старта приложения в момент, когда нет связи с Интернет.)
3. В текущей реализации приложения интуитивно не ясно его назначение. Возможно, вы знаете как сделать интерфейс более явным/привлекательным. Попробуйте добавить элементы/изменить расположение текущих для более удобной работы с приложением.