SMARTEX

Invest Smart

Lakshmi Kanth Sandra
229653990
Komal RajendraKumar Gujarathi
211778351

Table of Contents

Introduction

With the advent of globalisation and increasing foreign trade across the world there is a tremendous growth in the foreign exchange market. Billions of dollars gets exchanged every day in the bourses across the world.

Some traders use Technical analysis and others use fundamental analysis for FOREX trading purpose. Former technique mainly depends on quantitative analysis of the data where as the latter depends on the socio-politico-economic conditions of the countries across the world. This implies that there is no one fool proof strategy for effective decision making.

Several resources like newspapers and magazines help the investors by providing information that can help them make fundamental analysis. Added to that there are several quantitative tools in the market which aide investors in making effective investment decisions. They provide multiple analytical and visual presentations to the investor. Forexlive, Forex and XE currency are some of the web/mobile applications in this category. They provide rich and diverse amount of visual presentations. But this overwhelms the novel retail user. Reading multiple charts, comparing them and analysing them necessitates both knowledge and time from the user, which s/he might not be capable of.

To solve this issue our SMARTEx provides a simple interface in which even a naive user can make effective decisions in a conniving market. It gives him freedom to choose both the currency in which s/he wants to buy and the currency which s/he wants to sell. It democratises the decision making as the user is given both the expert recommendations as well as historical and live data, unlike the other apps.

Entering the Arena

Choosing Base Currency

Here the investor enters the currency of his choice. This is the currency which is used to buy other currencies. The user will enter the currency in three letter format. It is a ISO 4217 standard. S/he also has the freedom to touch select any of the currencies displayed in the list. The list is scrollable. Country flags along with 3 letter ISO codes further provide help to identify the currencies.

Choosing To Currency

Upon selecting the base currency and touching the search button next screen is displayed in which a list of currencies of with live conversion rate with respect to the base currency is provided. User can either scroll to see the available currencies and touch select Or can directly search in the given search bar. 3 letter ISO currency codes are being used. The data that is being displayed here is a live data being updated for every 60 minutes. Upon selecting any one currency the user can touch the currency flag to move to the historical data screen. In addition to that there is a recommendation clickable button below.

Visualise Historical Data

Once the user selected any to-currency in the above step s/he gets to see the chart plotted with historical data for that to-currency. This is a foreign exchange data, of past 1 month from the current day, for the to-currency with respect to the base currency. The chart provides a zoom-in & out interface to the user to visualise the data and make his/her own analysis.

Recommendations

Upon clicking the recommendation the user would be taken to a screen where the top 7 currencies will be displayed. If the user ain’t sure of which currencies to trade for, s/he can use this expert suggestions. They are provided based on the proven statistical metric of exchange rate differentials across the month. Trending mantra in the investment market is that best fluctuating currencies are the best currencies to invest in.

Model-View-Architecture

In the concept of MVC, the Model is where our data resides, the View is the face of our app which the user sees and the Controller mediates between the View and the Model. We have followed the same MVC approach.

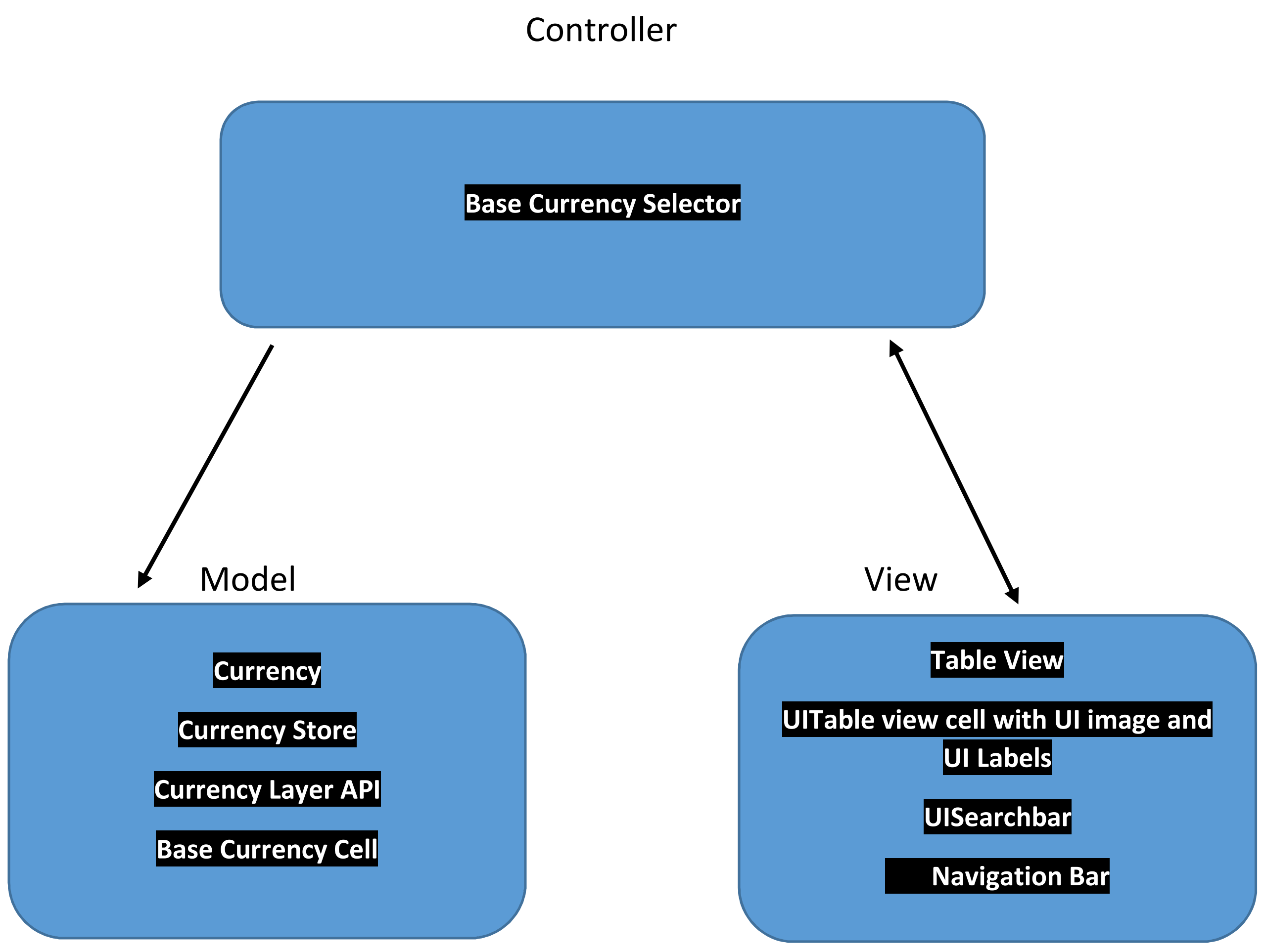


Fig-1 Base currency Selector

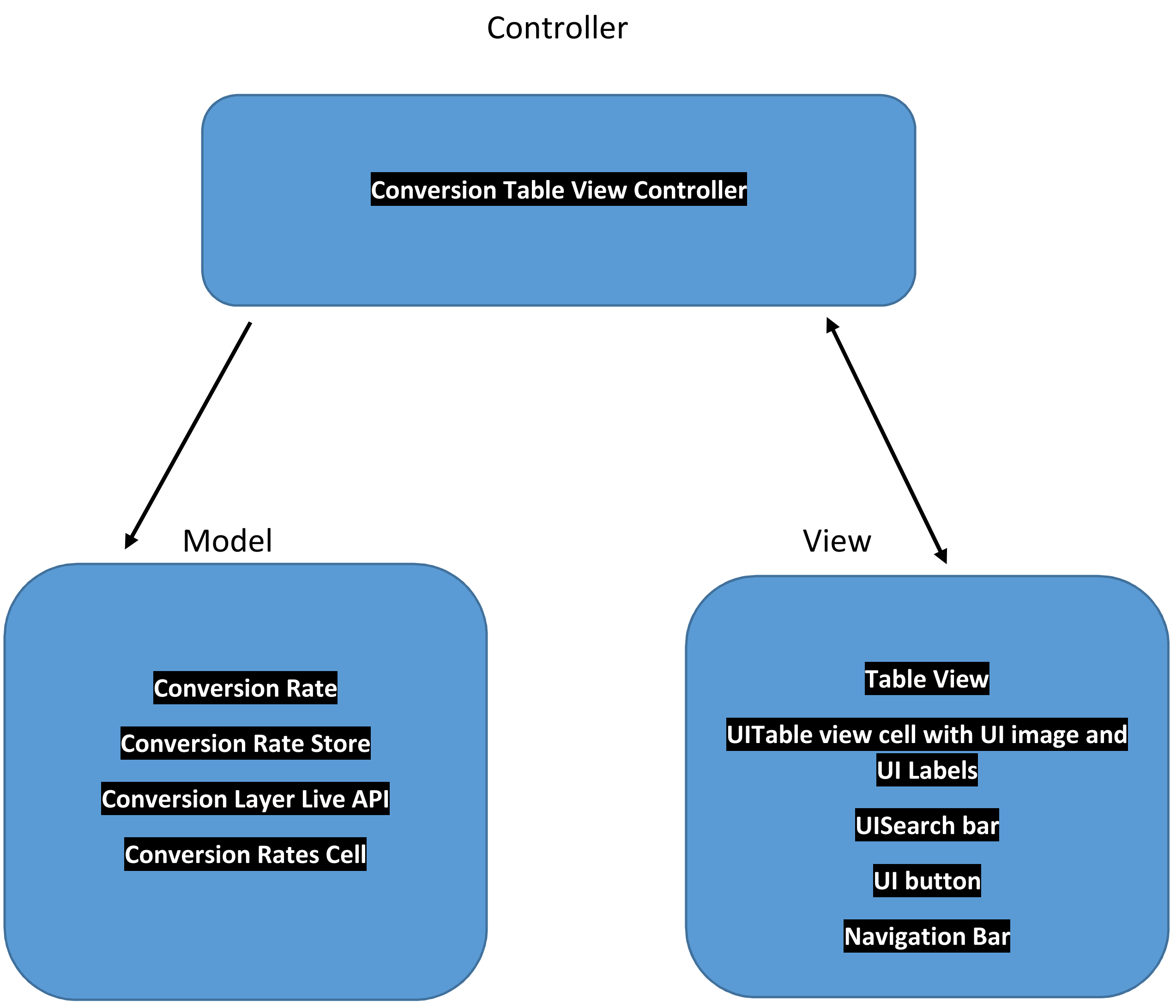


Fig-2 Live Conversion

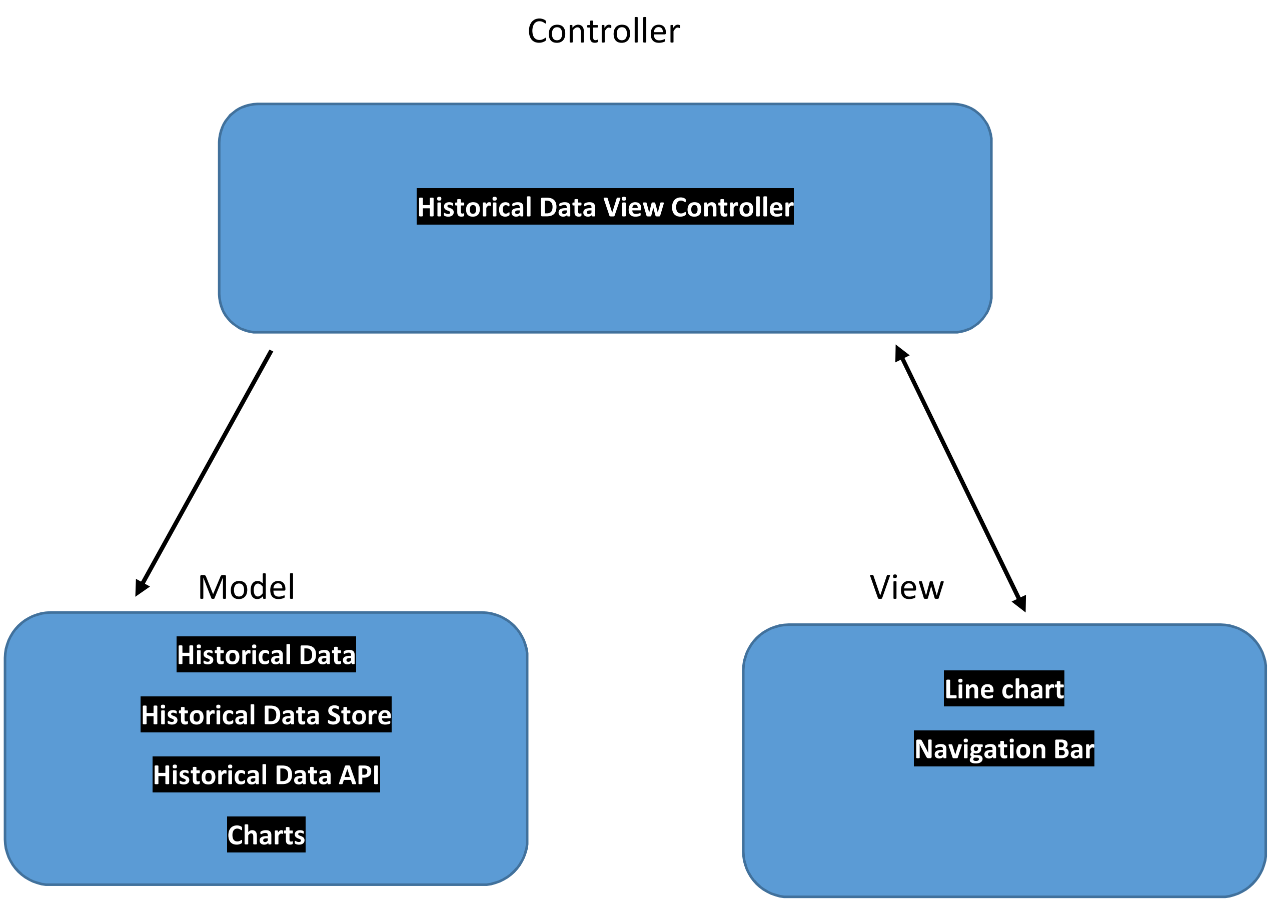


Fig- 3 Historical Data

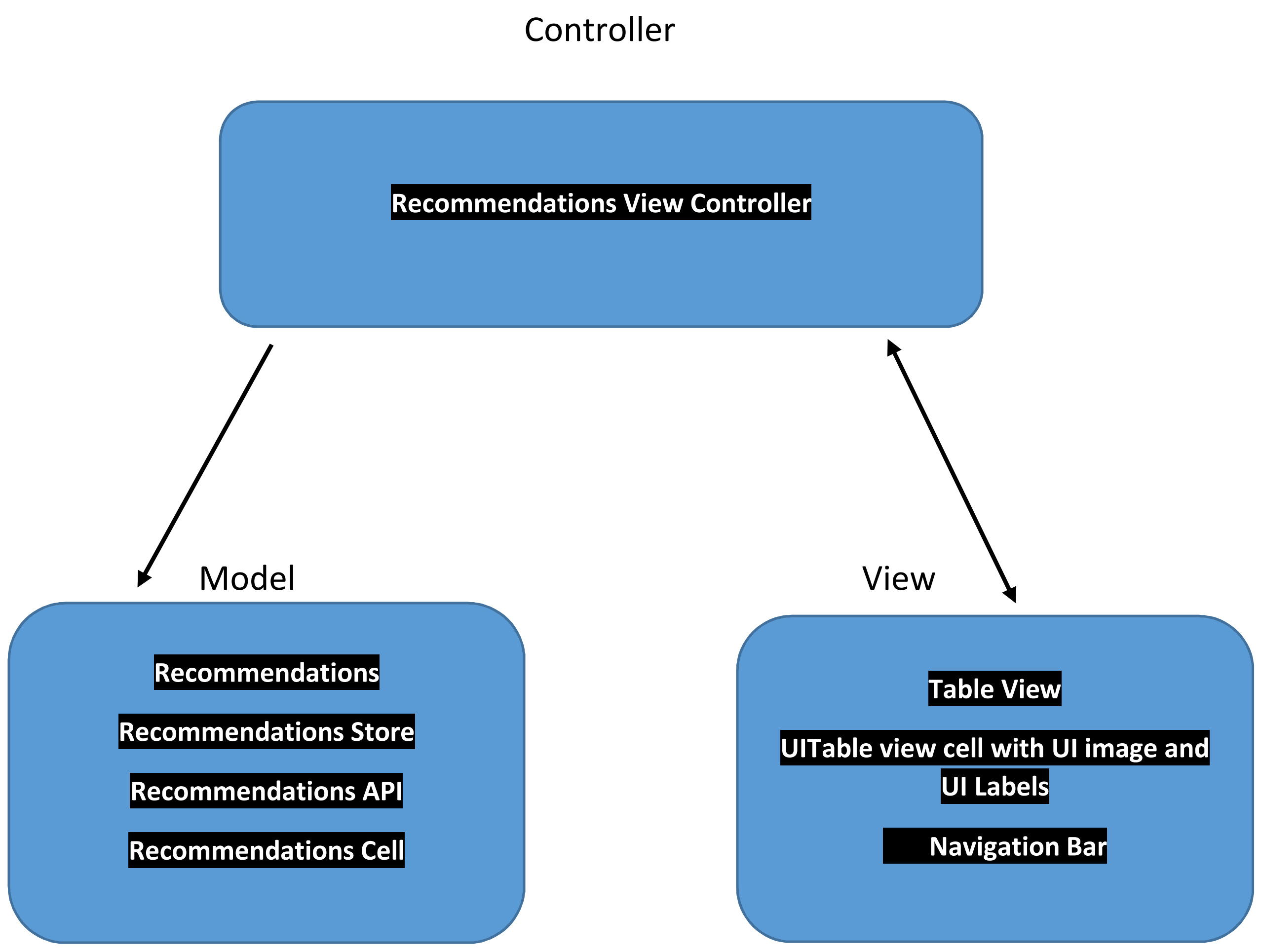


Fig-4 Recommendation system

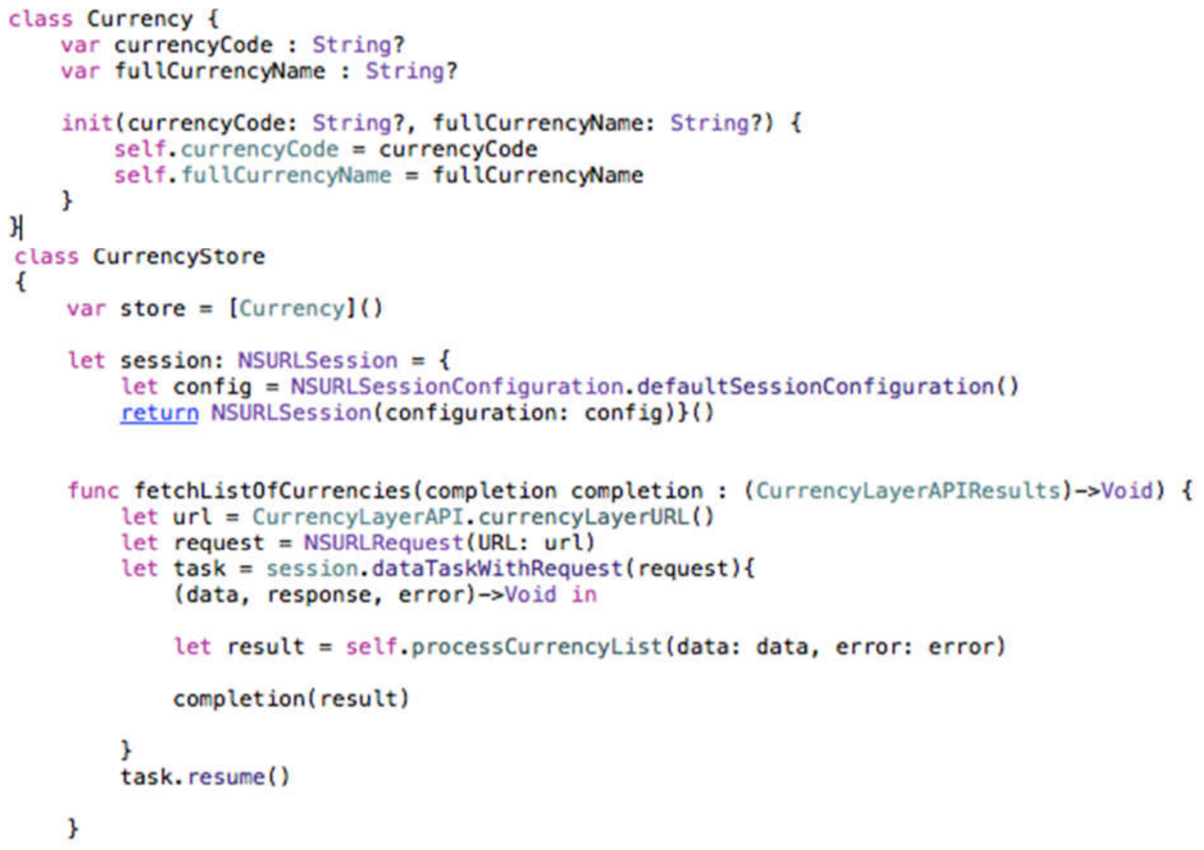
Design and Implementation

The project follows, mainly, the design pattern of single responsibility principle and Dependency Inversion Principle(DIP). There are four packages in the design.

1. Base currency selection
2. Conversion live rate
3. Historical Data
4. Recommendation system

In each package the low level design is segregated from its immediate higher level design which enforces DIP. To illustrate, the data variables are segregated in to a custom class. Eg- currency.swift, conversionrate.swift, historicaldata.swift, recommendations.swift

Fig- code snapshots of currency.swift and currencystore.swift



```swift
class Currency {
    var currencyCode : String?
    var fullCurrencyName : String?

    init(currencyCode: String?, fullCurrencyName: String?) {
        self.currencyCode = currencyCode
        self.fullCurrencyName = fullCurrencyName
    }
}
class CurrencyStore
{
    var store = [Currency]()

    let session: NSURLSession = {
        let config = NSURLSessionConfiguration.defaultSessionConfiguration()
        return NSURLSession(configuration: config)}()


    func fetchListOfCurrencies(completion completion : (CurrencyLayerAPIResults)->Void) {
        let url = CurrencyLayerAPI.currencyLayerURL()
        let request = NSURLRequest(URL: url)
        let task = session.dataTaskWithRequest(request){
            (data, response, error)->Void in

            let result = self.processCurrencyList(data: data, error: error)

            completion(result)

        }
        task.resume()

    }
```

A store class is created in each package in which there are methods that perform operations on data. This class also composes the API data class which generates specific

Package Structure

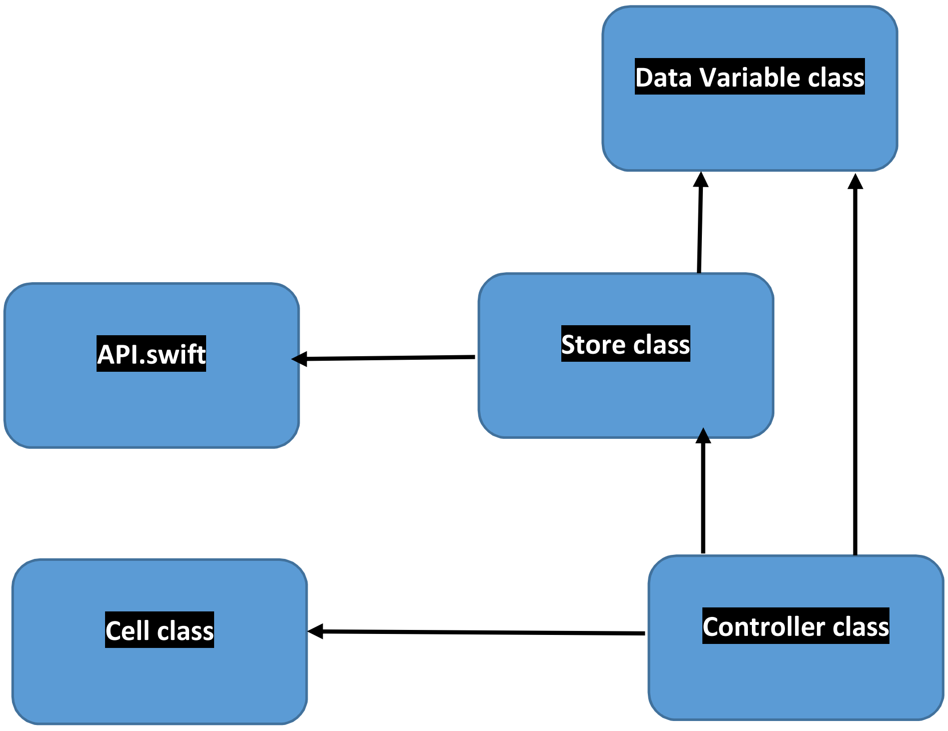
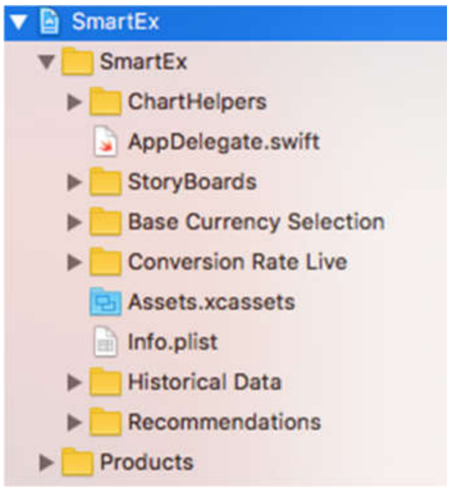


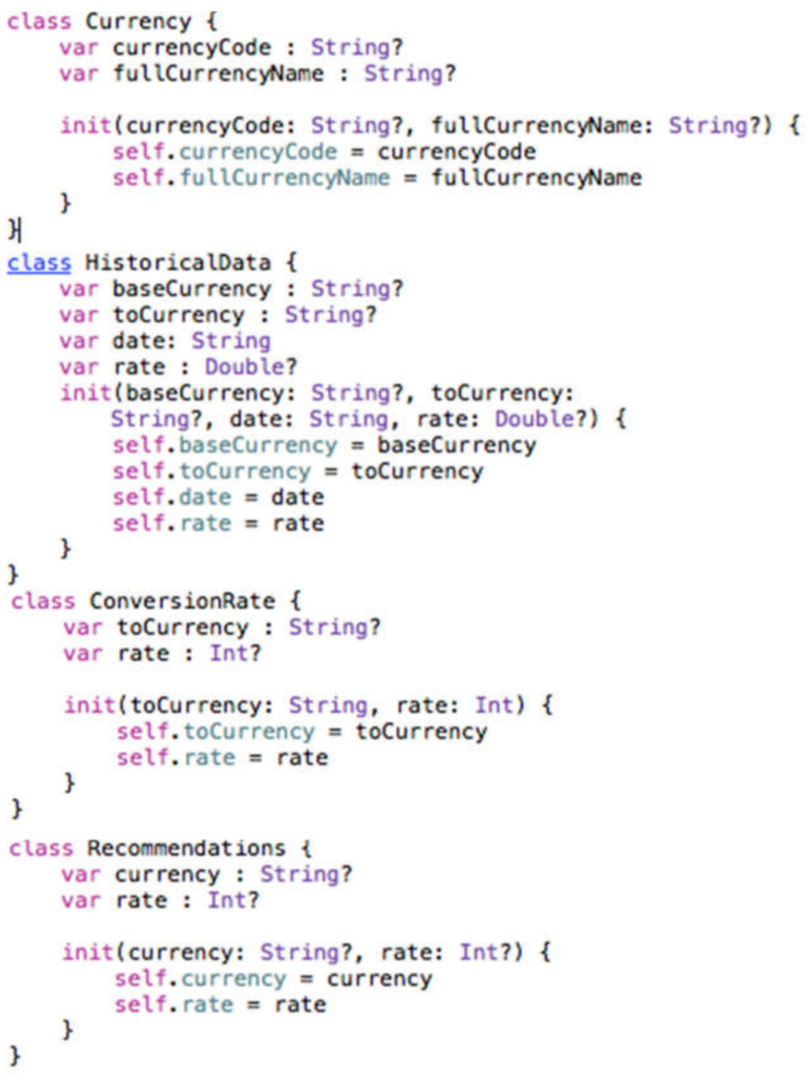
Fig-5 General package structure followed

Fig-6 snapshot of the packages


SmartEx
SmartEx
ChartHelpers
AppDelegate.swift
StoryBoards
Base Currency Selection
Conversion Rate Live
Assets.xcassets
Info.plist
Historical Data
Recommendations
Products

Data.swift

1. Low level data variables are segregated in to a custom class so that they can be easily extendable in future. The following snippets shows the data variable classes.
2. Fig- code snapshots of all data classes

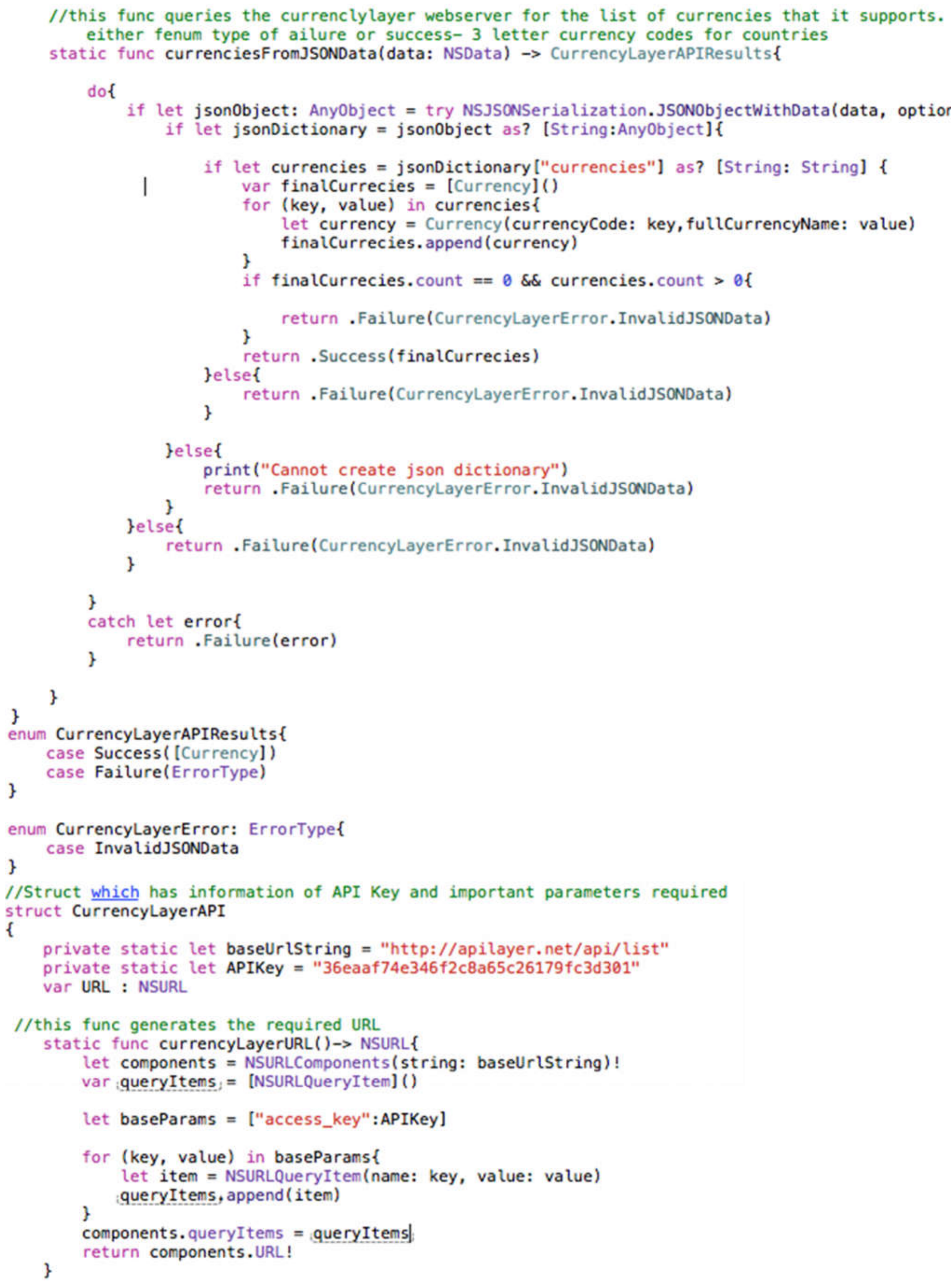


```
class Currency {
    var currencyCode : String?
    var fullCurrencyName : String?

    init(currencyCode: String?, fullCurrencyName: String?) {
        self.currencyCode = currencyCode
        self.fullCurrencyName = fullCurrencyName
    }
}
class HistoricalData {
    var baseCurrency : String?
    var toCurrency : String?
    var date: String
    var rate : Double?
    init(baseCurrency: String?, toCurrency:
        String?, date: String, rate: Double?) {
        self.baseCurrency = baseCurrency
        self.toCurrency = toCurrency
        self.date = date
        self.rate = rate
    }
}
class ConversionRate {
    var toCurrency : String?
    var rate : Int?

    init(toCurrency: String, rate: Int) {
        self.toCurrency = toCurrency
        self.rate = rate
    }
}

class Recommendations {
    var currency : String?
    var rate : Int?

    init(currency: String?, rate: Int?) {
        self.currency = currency
        self.rate = rate
    }
}
```

API.swift

1. API.swift has a struct which contains API key and URLs of the respective package as private data to prevent any untoward data corruption.
2. The struct segregates the base URL, API key and Query parameters. This ensures easy extension in the future.
3. There exists a static function in each API.swift file to generate the needed URL. This function is being accessed by the store class. The function returns an URL.
4. There exists a function to query data from the API server in JSON format. This function returns an enum type to consider the possibility that the result of the query might be successful or yield various types of errors. The Server might provide the functionality of sending type of error that occurred. Thus enum return type helps in processing it. This function is being accessed by the controller through store class.

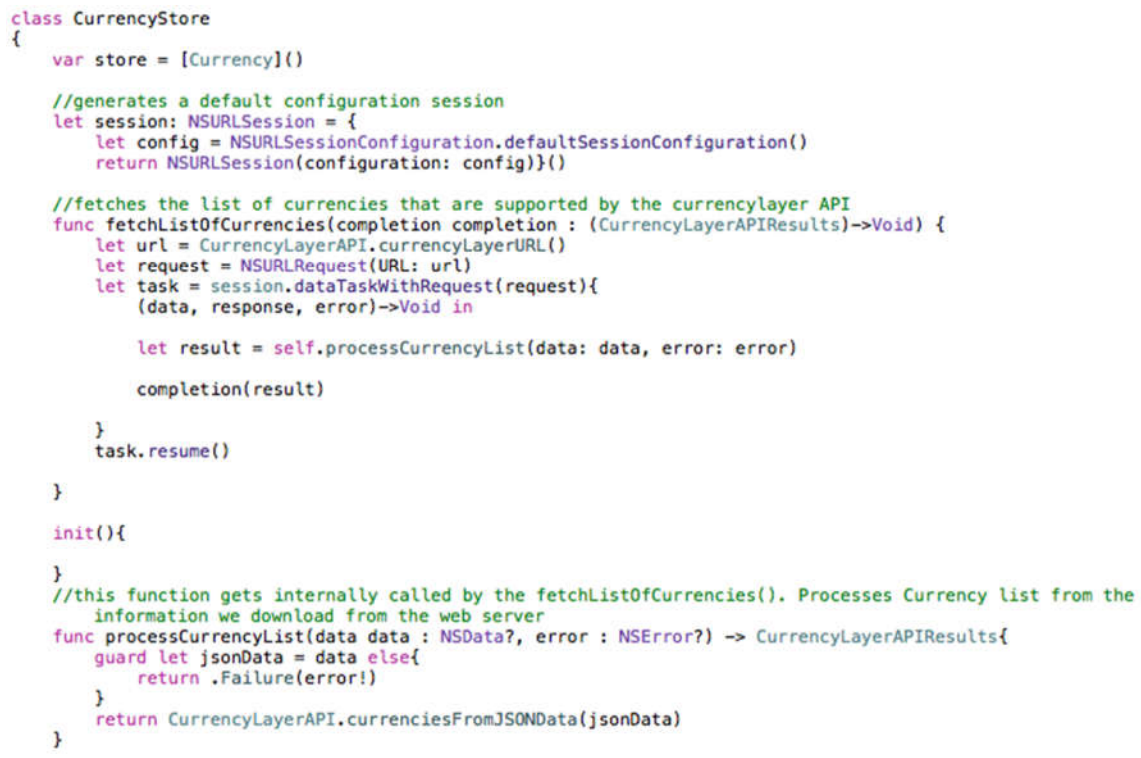
5. Fig-7 code snapshots of JSON query, enum types and URL fetching



```
    //this func queries the currenclylayer webserver for the list of currencies that it supports.
        either fenum type of ailure or success- 3 letter currency codes for countries
    static func currenciesFromJSONData(data: NSData) -> CurrencyLayerAPIResults{

        do{
            if let jsonObject: AnyObject = try NSJSONSerialization.JSONObjectWithData(data, optio
                if let jsonDictionary = jsonObject as? [String:AnyObject]{

                    if let currencies = jsonDictionary["currencies"] as? [String: String] {
                        var finalCurrecies = [Currency]()
                        for (key, value) in currencies{
                            let currency = Currency(currencyCode: key,fullCurrencyName: value)
                            finalCurrecies.append(currency)
                        }
                        if finalCurrecies.count == 0 && currencies.count > 0{

                            return .Failure(CurrencyLayerError.InvalidJSONData)
                        }
                        return .Success(finalCurrecies)
                    }else{
                        return .Failure(CurrencyLayerError.InvalidJSONData)
                    }

                }else{
                    print("Cannot create json dictionary")
                    return .Failure(CurrencyLayerError.InvalidJSONData)
                }
            }else{
                return .Failure(CurrencyLayerError.InvalidJSONData)
            }

        }
        catch let error{
            return .Failure(error)
        }

    }
}
enum CurrencyLayerAPIResults{
    case Success([Currency])
    case Failure(ErrorType)
}

enum CurrencyLayerError: ErrorType{
    case InvalidJSONData
}
//Struct which has information of API Key and important parameters required
struct CurrencyLayerAPI
{
    private static let baseUrlString = "http://apilayer.net/api/list"
    private static let APIKey = "36eaaf74e346f2c8a65c26179fc3d301"
    var URL : NSURL

 //this func generates the required URL
    static func currencyLayerURL()-> NSURL{
        let components = NSURLComponents(string: baseUrlString)!
        var queryItems = [NSURLQueryItem]()

        let baseParams = ["access_key":APIKey]

        for (key, value) in baseParams{
            let item = NSURLQueryItem(name: key, value: value)
            queryItems.append(item)
        }
        components.queryItems = queryItems
        return components.URL!
    }
```

Store.swift

1. The custom store class composes the data class.
2. Store class creates an interface function for the controller to call. This function internally uses the API.siwft for calling URL function and the Queryfunction.
3. A new session with default configuration is created by the store class using the NSURLSessionConfiguration.defaultsessionConfiguration(). This session is used by the above interface function to fire a query through session.dataTaskWIthRequest().
4. Fig-8 currencyStore class



```
class CurrencyStore
{
    var store = [Currency]()

    //generates a default configuration session
    let session: NSURLSession = {
        let config = NSURLSessionConfiguration.defaultSessionConfiguration()
        return NSURLSession(configuration: config)}()

    //fetches the list of currencies that are supported by the currencylayer API
    func fetchListOfCurrencies(completion completion : (CurrencyLayerAPIResults)->Void) {
        let url = CurrencyLayerAPI.currencyLayerURL()
        let request = NSURLRequest(URL: url)
        let task = session.dataTaskWithRequest(request){
            (data, response, error)->Void in

            let result = self.processCurrencyList(data: data, error: error)

            completion(result)

        }
        task.resume()

    }

    init(){

    }
    //this function gets internally called by the fetchListOfCurrencies(). Processes Currency list from the
        information we download from the web server
    func processCurrencyList(data data : NSData?, error : NSError?) -> CurrencyLayerAPIResults{
        guard let jsonData = data else{
            return .Failure(error!)
        }
        return CurrencyLayerAPI.currenciesFromJSONData(jsonData)
    }
```

Controller.swift

1. There are 5 controllers in the SMARTEx viz.
2. BaseCurrencySelector
3. ConversionTableVIewController
4. HistoricaldataViewController
5. RecommendationTableViewController
6. Navigation Controller

BaseCurrencySelector

1. The purpose of this controller is to instantiate the UITableview, process the user input string and display the base currency cell.
2. Thus the controller subclasses as follows

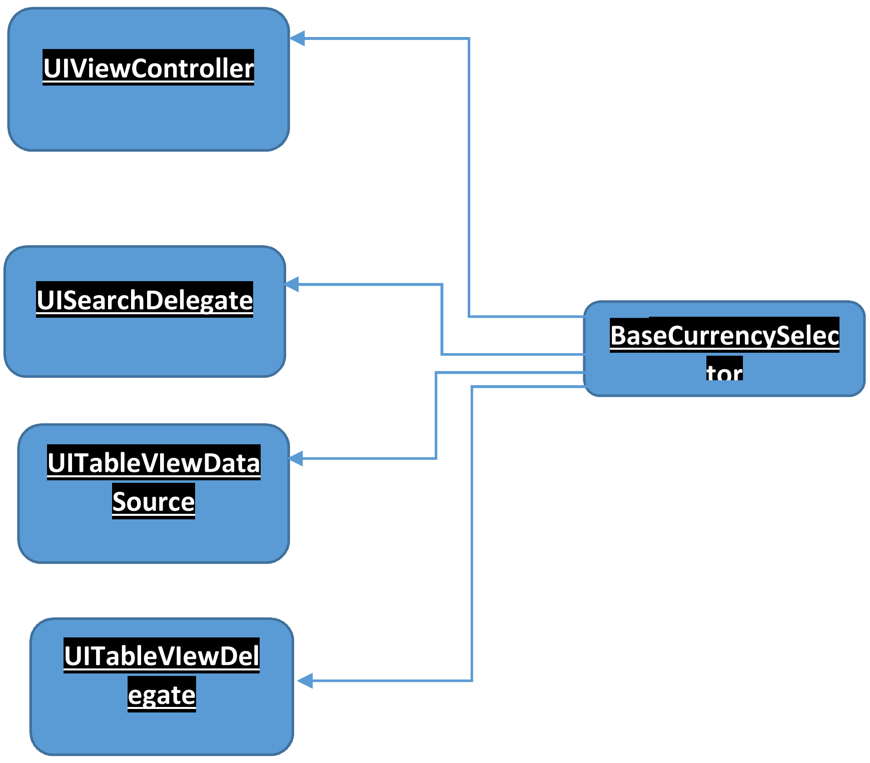
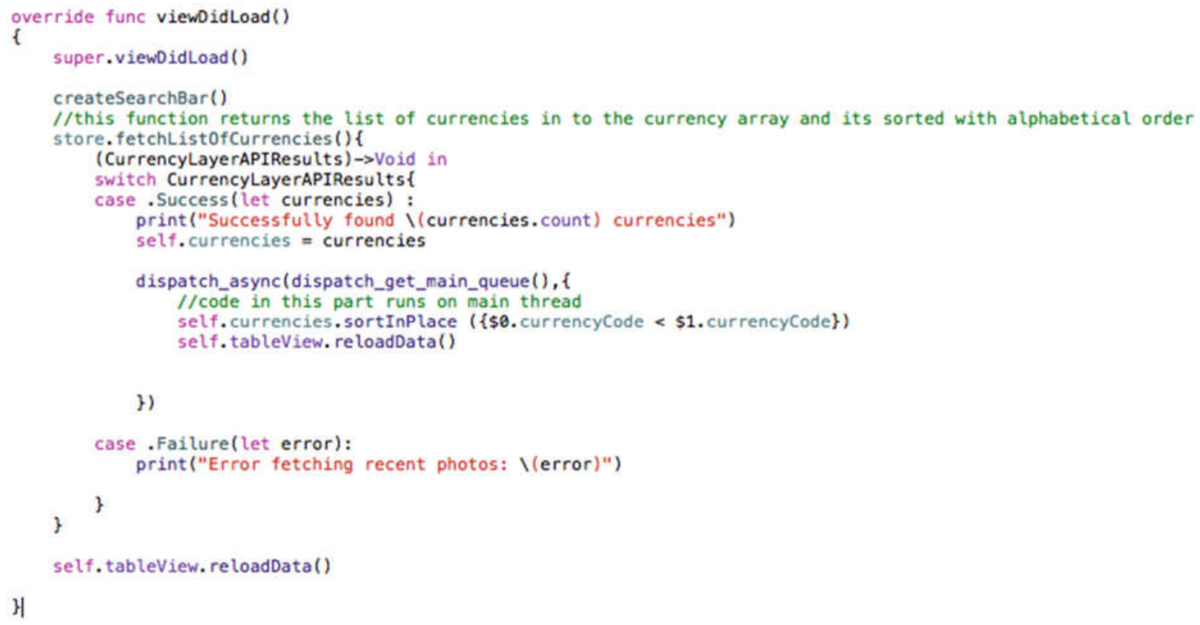


Fig-9 BaseCurrencySelector inheritance and conformity

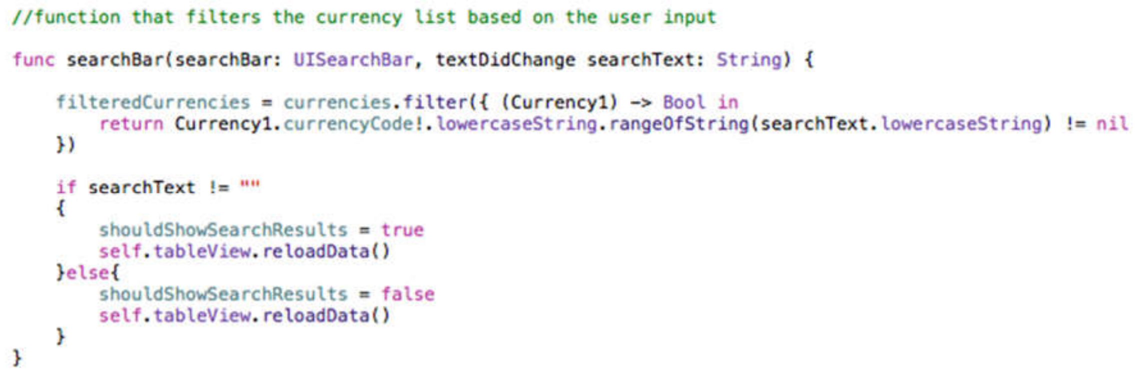
3. When the user inputs a text string it is processed and the number of rows in the tabaleview to be displayed are returned accordingly. The content of the rows will be as follows:
4. If user text matches the characters of the 3 letter currency code then only those filtered currencies will be displayed.

5. If the user text doesn't match any of the currencies in the list and its not an empty string then a no currency available cell will be displayed.
6. If the user text is empty then all the currencies returned by the API will be displayed.
7. Following are the code snippets that mainly perform this filtering
8. Fig –10 fetchingCurrencyList from webserver via currencystore class.



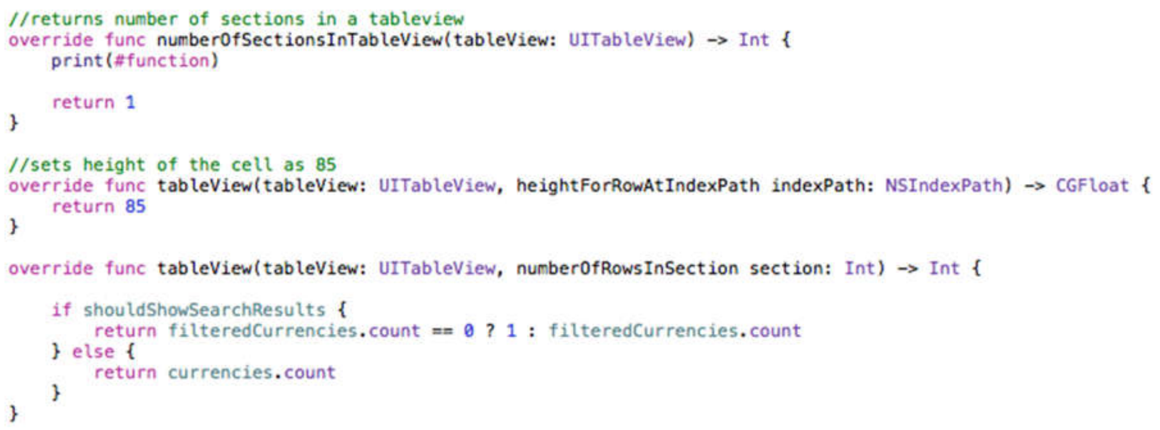
```
override func viewDidLoad()
{
    super.viewDidLoad()

    createSearchBar()
    //this function returns the list of currencies in to the currency array and its sorted with alphabetical order
    store.fetchListOfCurrencies(){
        (CurrencyLayerAPIResults)->Void in
        switch CurrencyLayerAPIResults{
        case .Success(let currencies) :
            print("Successfully found \(currencies.count) currencies")
            self.currencies = currencies

            dispatch_async(dispatch_get_main_queue(),{
                //code in this part runs on main thread
                self.currencies.sortInPlace ({$0.currencyCode < $1.currencyCode})
                self.tableView.reloadData()


            })

        case .Failure(let error):
            print("Error fetching recent photos: \(error)")

        }
    }

    self.tableView.reloadData()

}
```

9. Fig-11 search bar for processing user input.

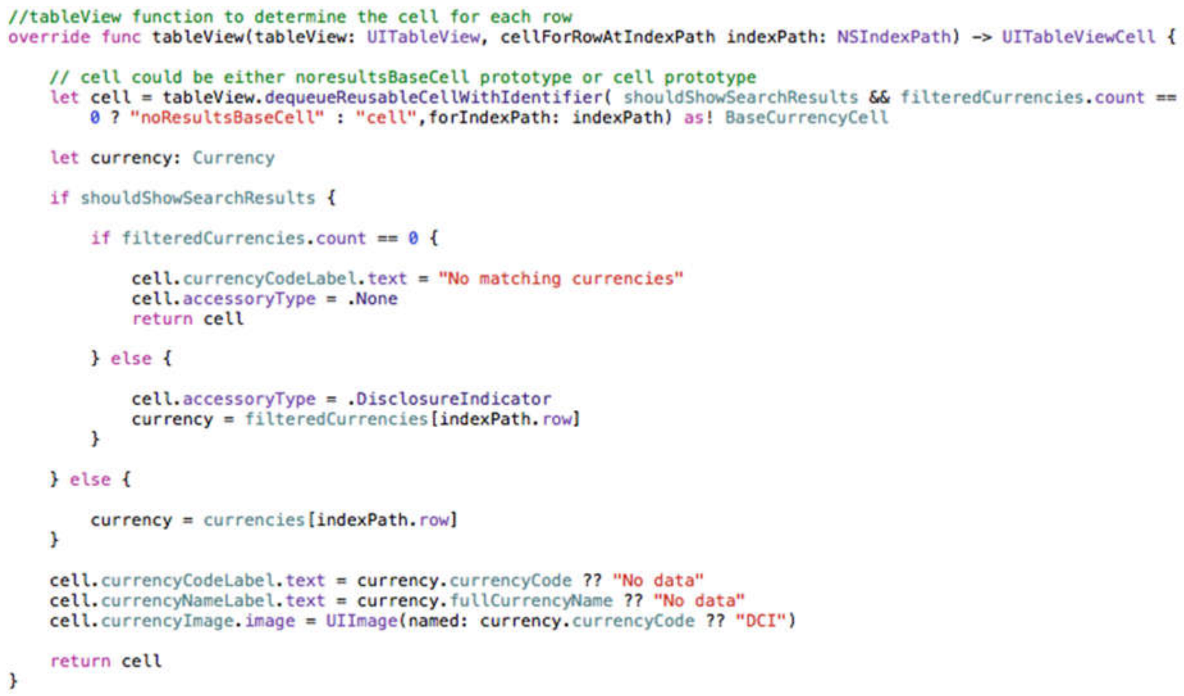


```
//function that filters the currency list based on the user input

func searchBar(searchBar: UISearchBar, textDidChange searchText: String) {

    filteredCurrencies = currencies.filter({ (Currency1) -> Bool in
        return Currency1.currencyCode!.lowercaseString.rangeOfString(searchText.lowercaseString) != nil
    })

    if searchText != ""
    {
        shouldShowSearchResults = true
        self.tableView.reloadData()
    }else{
        shouldShowSearchResults = false
        self.tableView.reloadData()
    }
}
```

10. Fig-12 tablewView instantiations in base currency selector



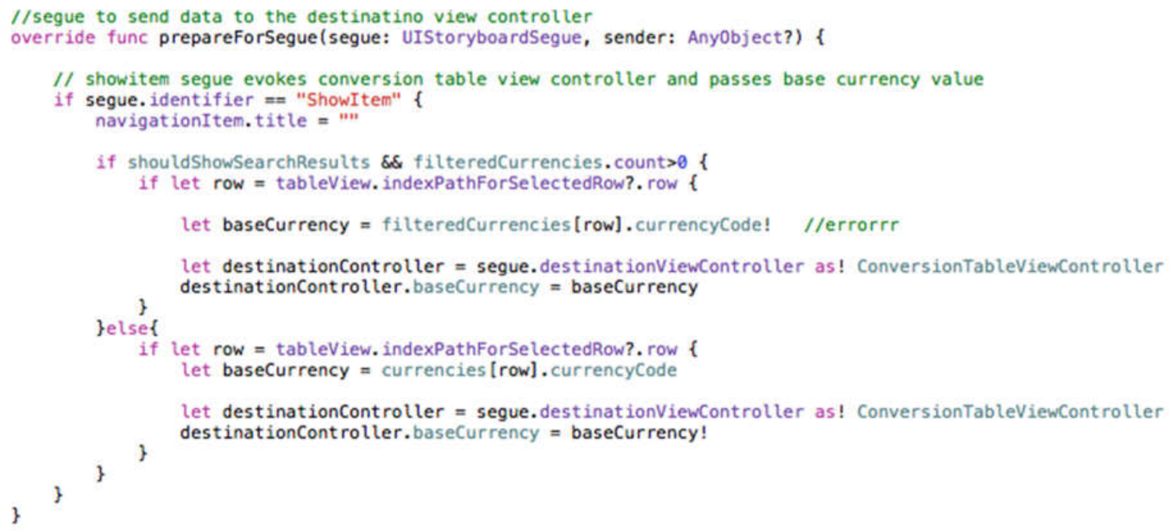
```
//returns number of sections in a tableview
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    print(#function)

    return 1
}

//sets height of the cell as 85
override func tableView(tableView: UITableView, heightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {
    return 85
}

override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

    if shouldShowSearchResults {
        return filteredCurrencies.count == 0 ? 1 : filteredCurrencies.count
    } else {
        return currencies.count
    }
}
```



```
//tableView function to determine the cell for each row
override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {

    // cell could be either noresultsBaseCell prototype or cell prototype
    let cell = tableView.dequeueReusableCellWithIdentifier( shouldShowSearchResults && filteredCurrencies.count ==
        0 ? "noResultsBaseCell" : "cell",forIndexPath: indexPath) as! BaseCurrencyCell

    let currency: Currency

    if shouldShowSearchResults {

        if filteredCurrencies.count == 0 {

            cell.currencyCodeLabel.text = "No matching currencies"
            cell.accessoryType = .None
            return cell

        } else {

            cell.accessoryType = .DisclosureIndicator
            currency = filteredCurrencies[indexPath.row]
        }

    } else {

        currency = currencies[indexPath.row]
    }

    cell.currencyCodeLabel.text = currency.currencyCode ?? "No data"
    cell.currencyNameLabel.text = currency.fullCurrencyName ?? "No data"
    cell.currencyImage.image = UIImage(named: currency.currencyCode ?? "DCI")

    return cell
}
```

11.Segue, Show Item, performs the transition between BasecurrencySelector and the conversionTablevIewController. It passes the base currency data, that has been selected by the user, from the basecurrencyselector to conversionViewTable controller.

12. Fig-13 segue from base currency controller to conversion view controller

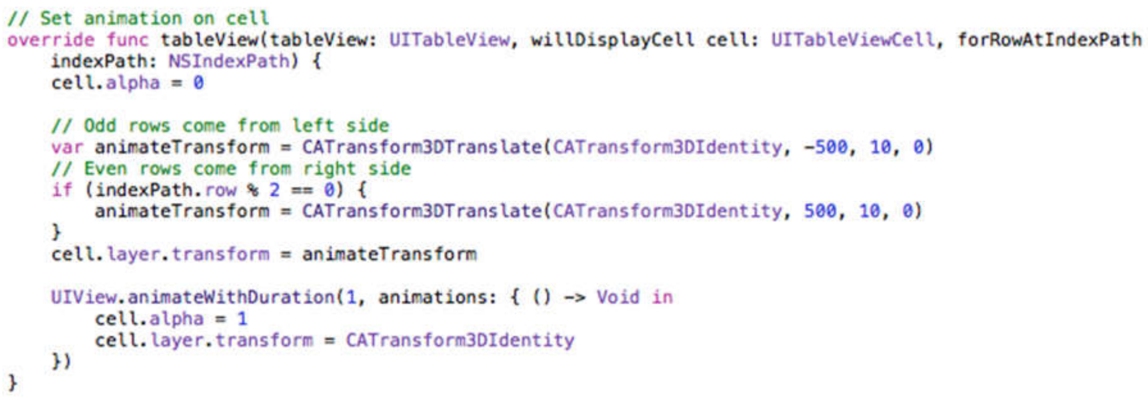


```
//segue to send data to the destinatino view controller
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {

    // showitem segue evokes conversion table view controller and passes base currency value
    if segue.identifier == "ShowItem" {
        navigationItem.title = ""

        if shouldShowSearchResults && filteredCurrencies.count>0 {
            if let row = tableView.indexPathForSelectedRow?.row {

                let baseCurrency = filteredCurrencies[row].currencyCode!   //errorrr

                let destinationController = segue.destinationViewController as! ConversionTableViewController
                destinationController.baseCurrency = baseCurrency
            }
        }else{
            if let row = tableView.indexPathForSelectedRow?.row {
                let baseCurrency = currencies[row].currencyCode

                let destinationController = segue.destinationViewController as! ConversionTableViewController
                destinationController.baseCurrency = baseCurrency!
            }
        }
    }
}
```

13.

The tableview cells are customised to be displayed in the animation mode. Following function has been utilised to this purpose.

14. Fig-14 Animating UITableViewcells



```
// Set animation on cell
override func tableView(tableView: UITableView, willDisplayCell cell: UITableViewCell, forRowAtIndexPath
    indexPath: NSIndexPath) {
    cell.alpha = 0

    // Odd rows come from left side
    var animateTransform = CATransform3DTranslate(CATransform3DIdentity, -500, 10, 0)
    // Even rows come from right side
    if (indexPath.row % 2 == 0) {
        animateTransform = CATransform3DTranslate(CATransform3DIdentity, 500, 10, 0)
    }
    cell.layer.transform = animateTransform

    UIView.animateWithDuration(1, animations: { () -> Void in
        cell.alpha = 1
        cell.layer.transform = CATransform3DIdentity
    })
}
```

ConversionViewTableController.

1. The main functionality of this controller is to enable the user to pick any currency which s/he wants to buy, to show the live conversion rate for the selected to-currency and to prepare the segue for Historical Data controller as well as Recommendation system.
2. Thus the controller subclasses as follows:

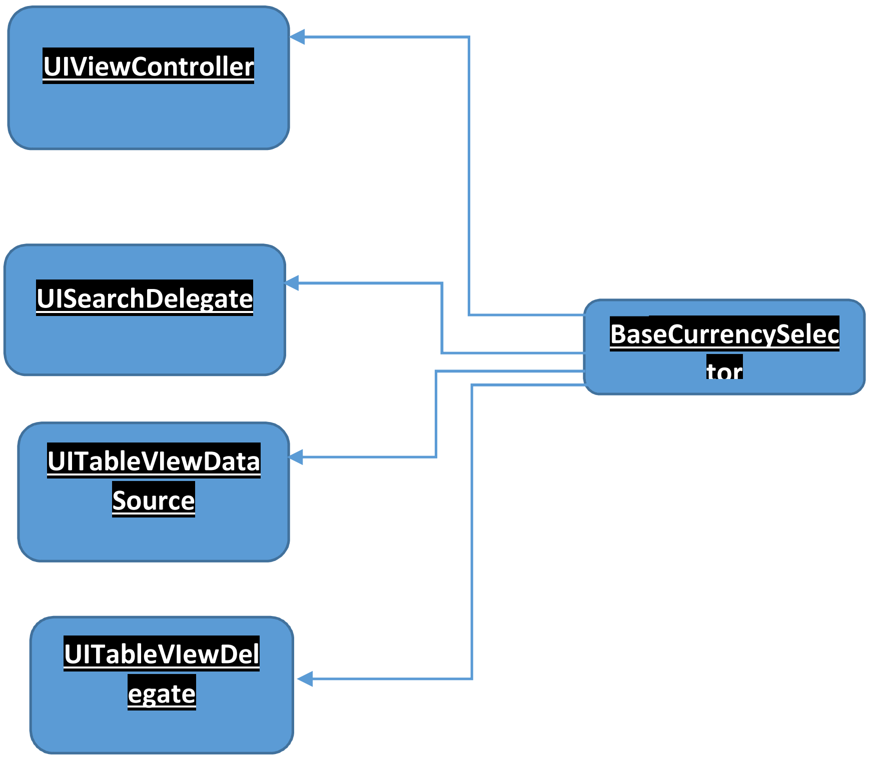
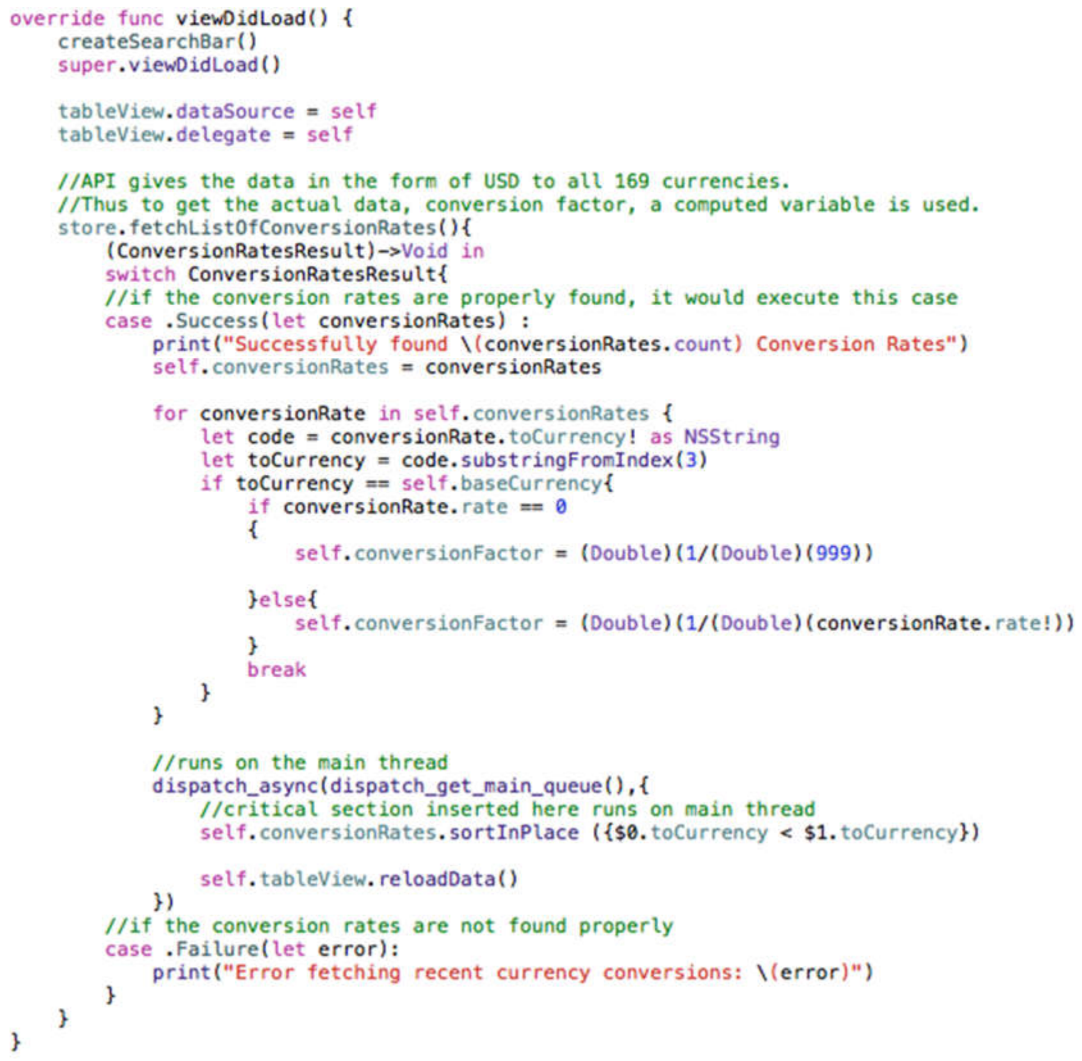


Fig-15 ConversionTableViewController inheritance and protocol conformity.

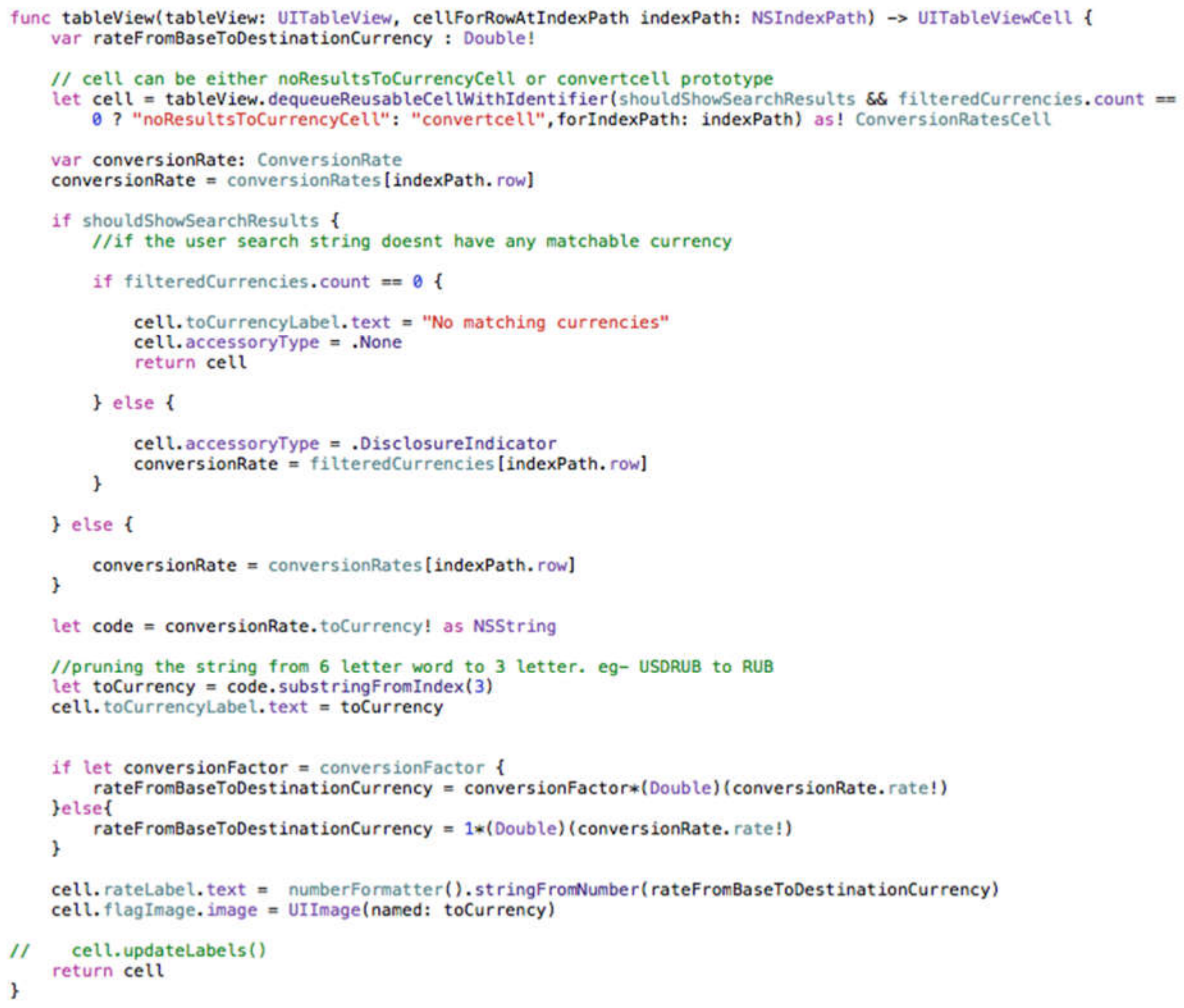
3. Apart from the search function and preparing the tableview, similar to the way basecurrencyselector did, this controller has to calculate the exchange rate between the user selected base currency and user selected to-currency. The challenge involved in the above step is that the currencyLayer API only provides exchange rate with USD as base currency. So the solution which we adopted was to calculate exchange rate as a computed property. conversionFactor variable is introduced for this purpose.

4. Fig –16 fetching live exchange rates from the web server.



```
override func viewDidLoad() {
    createSearchBar()
    super.viewDidLoad()

    tableView.dataSource = self
    tableView.delegate = self

    //API gives the data in the form of USD to all 169 currencies.
    //Thus to get the actual data, conversion factor, a computed variable is used.
    store.fetchListOfConversionRates(){
        (ConversionRatesResult)->Void in
        switch ConversionRatesResult{
        //if the conversion rates are properly found, it would execute this case
        case .Success(let conversionRates) :
            print("Successfully found \(conversionRates.count) Conversion Rates")
            self.conversionRates = conversionRates

            for conversionRate in self.conversionRates {
                let code = conversionRate.toCurrency! as NSString
                let toCurrency = code.substringFromIndex(3)
                if toCurrency == self.baseCurrency{
                    if conversionRate.rate == 0
                    {
                        self.conversionFactor = (Double)(1/(Double)(999))

                    }else{
                        self.conversionFactor = (Double)(1/(Double)(conversionRate.rate!))
                    }
                    break
                }
            }

            //runs on the main thread
            dispatch_async(dispatch_get_main_queue(),{
                //critical section inserted here runs on main thread
                self.conversionRates.sortInPlace ({$0.toCurrency < $1.toCurrency})

                self.tableView.reloadData()
            })
        //if the conversion rates are not found properly
        case .Failure(let error):
            print("Error fetching recent currency conversions: \(error)")
        }
    }
}
```

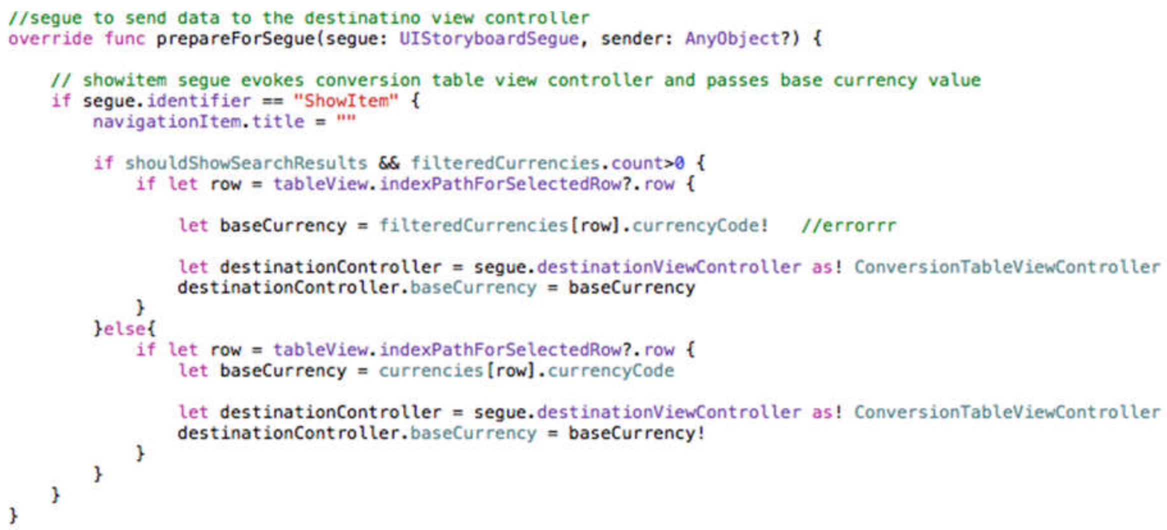
5. Fig-17 TableviewCells instantiation



```
func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    var rateFromBaseToDestinationCurrency : Double!

    // cell can be either noResultsToCurrencyCell or convertcell prototype
    let cell = tableView.dequeueReusableCellWithIdentifier(shouldShowSearchResults && filteredCurrencies.count ==
        0 ? "noResultsToCurrencyCell": "convertcell",forIndexPath: indexPath) as! ConversionRatesCell

    var conversionRate: ConversionRate
    conversionRate = conversionRates[indexPath.row]

    if shouldShowSearchResults {
        //if the user search string doesnt have any matchable currency

        if filteredCurrencies.count == 0 {

            cell.toCurrencyLabel.text = "No matching currencies"
            cell.accessoryType = .None
            return cell

        } else {

            cell.accessoryType = .DisclosureIndicator
            conversionRate = filteredCurrencies[indexPath.row]
        }

    } else {

        conversionRate = conversionRates[indexPath.row]
    }

    let code = conversionRate.toCurrency! as NSString

    //pruning the string from 6 letter word to 3 letter. eg- USDRUB to RUB
    let toCurrency = code.substringFromIndex(3)
    cell.toCurrencyLabel.text = toCurrency


    if let conversionFactor = conversionFactor {
        rateFromBaseToDestinationCurrency = conversionFactor*(Double)(conversionRate.rate!)
    }else{
        rateFromBaseToDestinationCurrency = 1*(Double)(conversionRate.rate!)
    }

    cell.rateLabel.text =  numberFormatter().stringFromNumber(rateFromBaseToDestinationCurrency)
    cell.flagImage.image = UIImage(named: toCurrency)

//     cell.updateLabels()
    return cell
}
```

6. This controller instantiates two segues viz. showDetail, showRecommendations. The former segue passes the basecurrency and to-currency variable data to the HistoricalDataViewController to enable plotting of the data. The latter segue doesn't pass any data apart from enabling the transition.

7. Fig-18 segue evoking either Historical controller or Recommendation

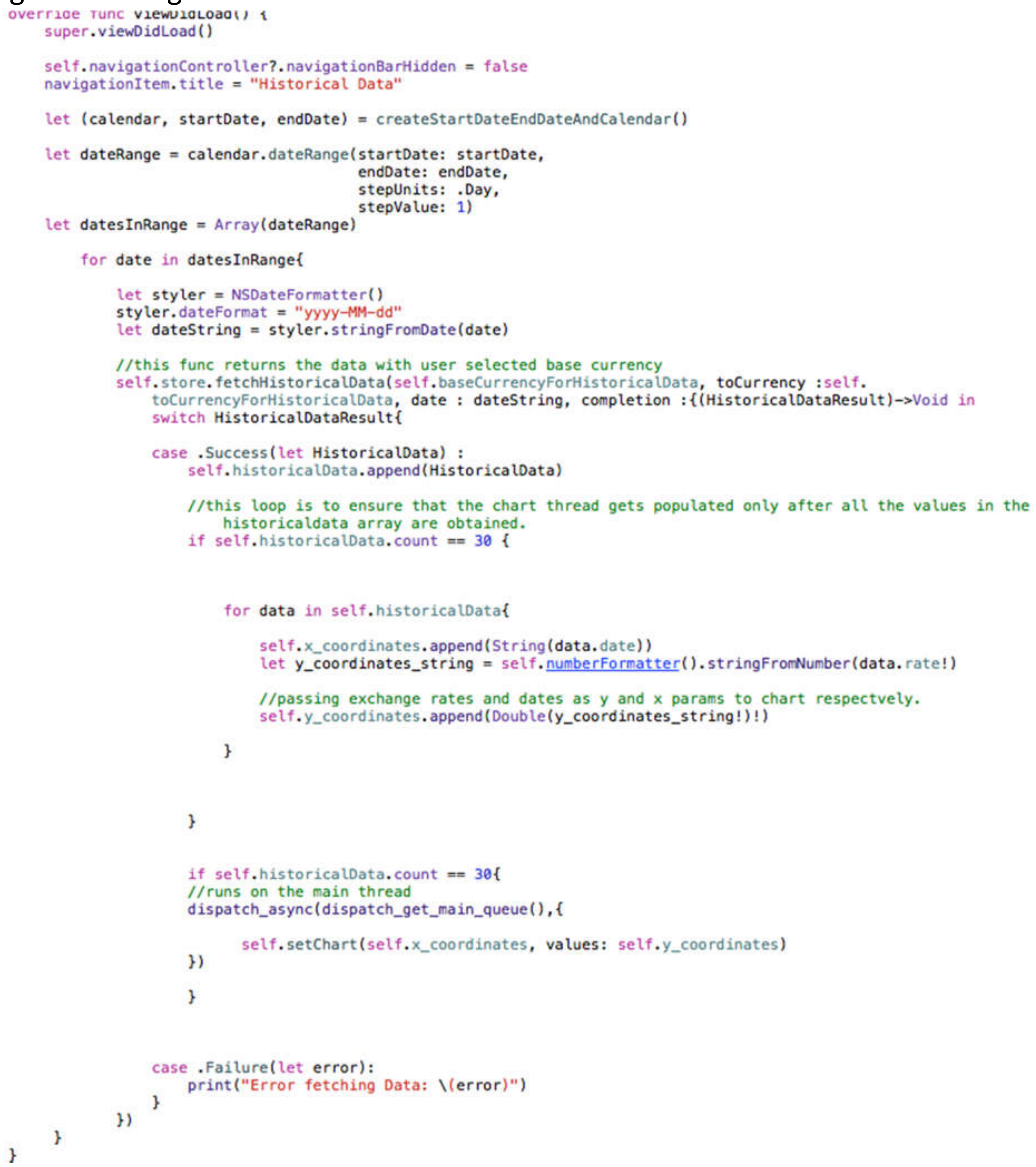


```
//segue to send data to the destinatino view controller
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {

    // showitem segue evokes conversion table view controller and passes base currency value
    if segue.identifier == "ShowItem" {
        navigationItem.title = ""

        if shouldShowSearchResults && filteredCurrencies.count>0 {
            if let row = tableView.indexPathForSelectedRow?.row {

                let baseCurrency = filteredCurrencies[row].currencyCode!    //errorrr

                let destinationController = segue.destinationViewController as! ConversionTableViewController
                destinationController.baseCurrency = baseCurrency
            }
        }else{
            if let row = tableView.indexPathForSelectedRow?.row {
                let baseCurrency = currencies[row].currencyCode

                let destinationController = segue.destinationViewController as! ConversionTableViewController
                destinationController.baseCurrency = baseCurrency!
            }
        }
    }
}
```

HistoricalDataViewController

1. This controller’s main functionality is to initiate the query for historical data and calculate it for the baseCurrency and to-Currency that has been obtained through showDetail segue.
2. This controller thus inherits from the UIViewcontroller and uses the Chart API.
3. The historical data is for the 30 days range starting from the current date.
4. An external public API for charts is integrated in this project. It takes the historical exchange rates and the dates as parameters for the y and x coordinates respectively.

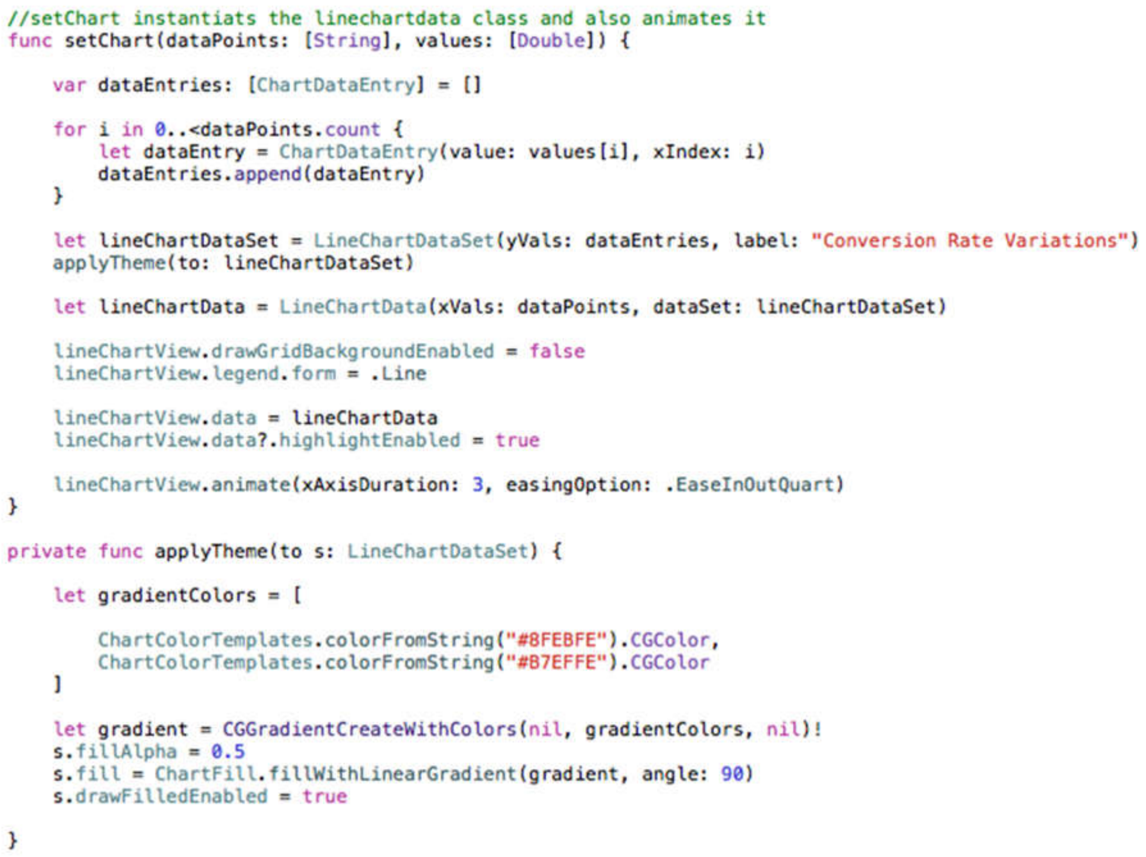
5. Fig-19 fetching historical data



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

    self.navigationController?.navigationBarHidden = false
    navigationItem.title = "Historical Data"

    let (calendar, startDate, endDate) = createStartDateEndDateAndCalendar()

    let dateRange = calendar.dateRange(startDate: startDate,
                                   endDate: endDate,
                                   stepUnits: .Day,
                                   stepValue: 1)
    let datesInRange = Array(dateRange)

        for date in datesInRange{

            let styler = NSDateFormatter()
            styler.dateFormat = "yyyy-MM-dd"
            let dateString = styler.stringFromDate(date)

            //this func returns the data with user selected base currency
            self.store.fetchHistoricalData(self.baseCurrencyForHistoricalData, toCurrency :self.
                toCurrencyForHistoricalData, date : dateString, completion :{(HistoricalDataResult)->Void in
                switch HistoricalDataResult{

                case .Success(let HistoricalData) :
                    self.historicalData.append(HistoricalData)

                    //this loop is to ensure that the chart thread gets populated only after all the values in the
                        historicaldata array are obtained.
                    if self.historicalData.count == 30 {


                        for data in self.historicalData{

                            self.x_coordinates.append(String(data.date))
                            let y_coordinates_string = self.numberFormatter().stringFromNumber(data.rate!)

                            //passing exchange rates and dates as y and x params to chart respectvely.
                            self.y_coordinates.append(Double(y_coordinates_string!)!)

                        }


                    }


                    if self.historicalData.count == 30{
                    //runs on the main thread
                    dispatch_async(dispatch_get_main_queue(),{

                          self.setChart(self.x_coordinates, values: self.y_coordinates)
                    })

                    }


                case .Failure(let error):
                    print("Error fetching Data: \(error)")
                }
            })
        }
}
```

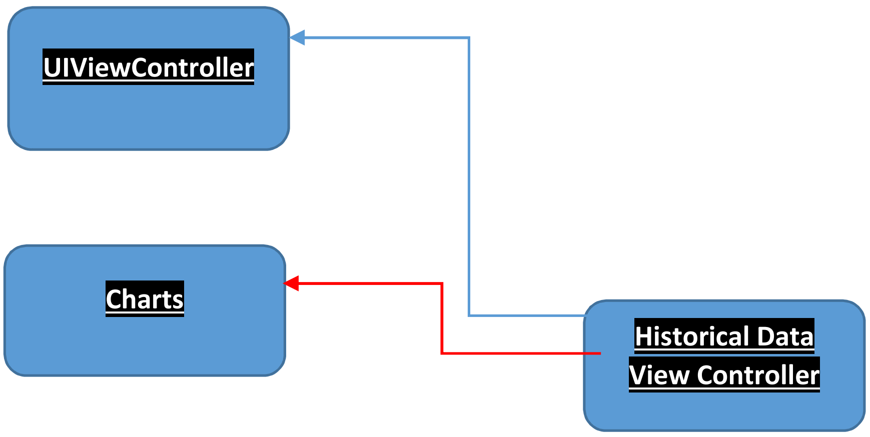
6. The chart is beautified for the visual appeal. Its filled with the background appeasing sky blue color. Its animated. It has a zoom in/out feature for the user ease.

7. Fig-20 chart plotted with historical data of exchange rates



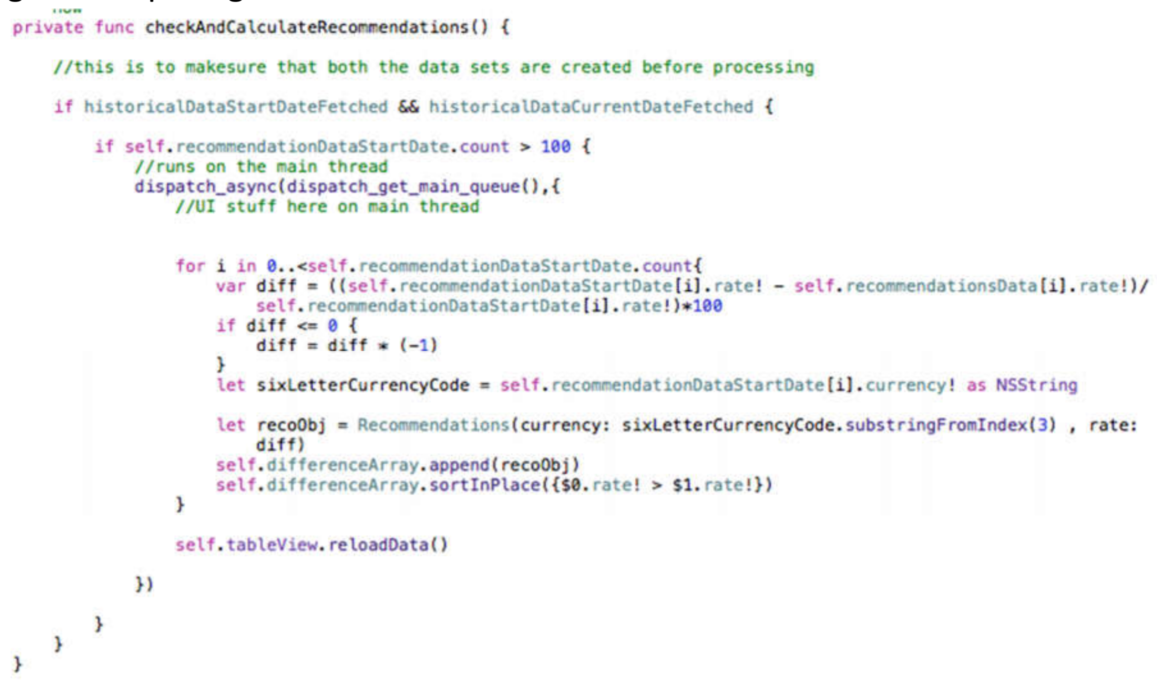
```
//setChart instantiats the linechartdata class and also animates it
func setChart(dataPoints: [String], values: [Double]) {

    var dataEntries: [ChartDataEntry] = []

    for i in 0..<dataPoints.count {
        let dataEntry = ChartDataEntry(value: values[i], xIndex: i)
        dataEntries.append(dataEntry)
    }

    let lineChartDataSet = LineChartDataSet(yVals: dataEntries, label: "Conversion Rate Variations")
    applyTheme(to: lineChartDataSet)

    let lineChartData = LineChartData(xVals: dataPoints, dataSet: lineChartDataSet)

    lineChartView.drawGridBackgroundEnabled = false
    lineChartView.legend.form = .Line

    lineChartView.data = lineChartData
    lineChartView.data?.highlightEnabled = true

    lineChartView.animate(xAxisDuration: 3, easingOption: .EaseInOutQuart)
}

private func applyTheme(to s: LineChartDataSet) {

    let gradientColors = [

        ChartColorTemplates.colorFromString("#8FEBFE").CGColor,
        ChartColorTemplates.colorFromString("#B7EFFE").CGColor
    ]

    let gradient = CGGradientCreateWithColors(nil, gradientColors, nil)!
    s.fillAlpha = 0.5
    s.fill = ChartFill.fillWithLinearGradient(gradient, angle: 90)
    s.drawFilledEnabled = true

}
```

8. Fig-21 Historical Data View controller class diagram.



RecommendationsTableViewController

1. Major functionality of this controller is to analyse the safe currencies which are relatively risk free to invest and display those currencies on the tableview
2. For this purpose we use the statistical metric of monthly differential and compute the top 7 currencies to display.
3. The controller directly subclasses the UITableviewcontroller. The UIView object that evokes this scene is located in the 2nd scene with title as Recommended Currencies.
4. The dates are computed similar to the HistoricalDataViewController using NSDate and NSCalendar classes.
5. Fig-22 computing the statistical differential metric



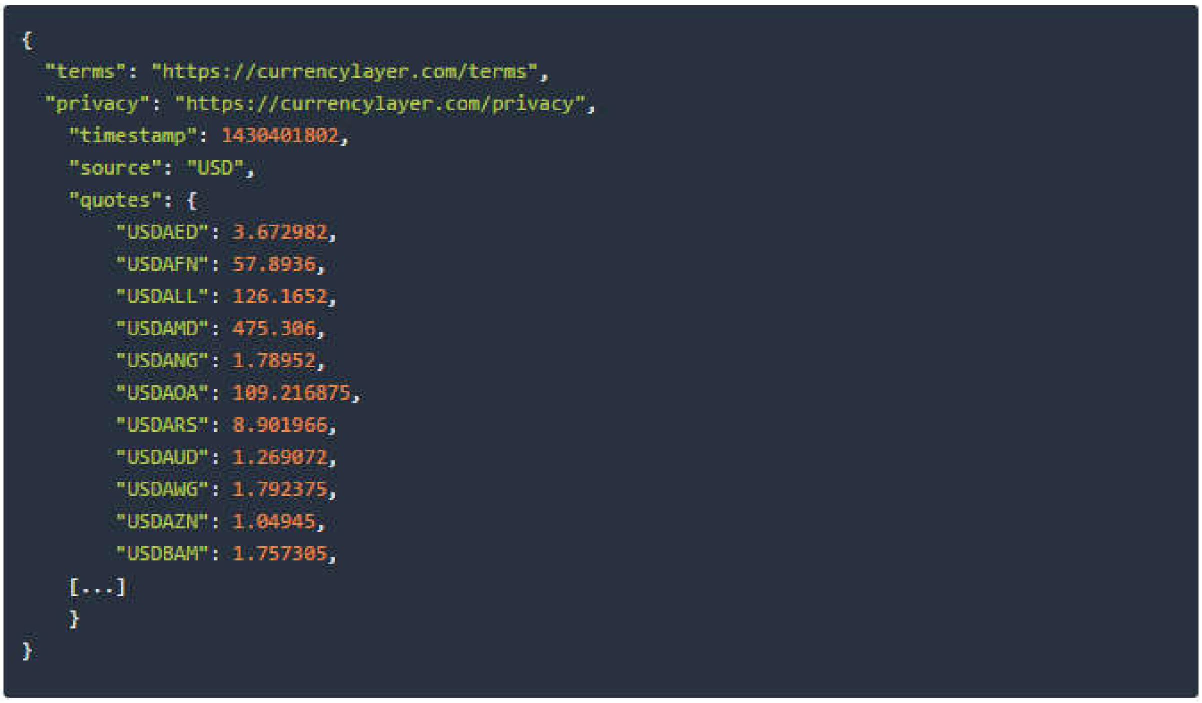
```
private func checkAndCalculateRecommendations() {

    //this is to makesure that both the data sets are created before processing

    if historicalDataStartDateFetched && historicalDataCurrentDateFetched {

        if self.recommendationDataStartDate.count > 100 {
            //runs on the main thread
            dispatch_async(dispatch_get_main_queue(),{
                //UI stuff here on main thread


                for i in 0..<self.recommendationDataStartDate.count{
                    var diff = ((self.recommendationDataStartDate[i].rate! - self.recommendationsData[i].rate!)/
                        self.recommendationDataStartDate[i].rate!)*100
                    if diff <= 0 {
                        diff = diff * (-1)
                    }
                    let sixLetterCurrencyCode = self.recommendationDataStartDate[i].currency! as NSString

                    let recoObj = Recommendations(currency: sixLetterCurrencyCode.substringFromIndex(3) , rate:
                        diff)
                    self.differenceArray.append(recoObj)
                    self.differenceArray.sortInPlace({$0.rate! > $1.rate!})
                }

                self.tableView.reloadData()

            })

        }
    }
}
```

Navigation Controller

1. The navigation controller is the initial view controller which embeds all the above controllers. Its top view controller initially will be the BasecurrencySelector.

API Integrated

1. CurrencyLayer API is the one which the project uses. This API is reputed for its accuracy, low latency and reliability of the data.
2. it provides the following data
   a. List of currencies supported
   b. Exchange rates with USD as base currency.
   c. Historical data for any given day.
3. The query result is in the JSON format. Following is the snapshot of the sample data format that it provides.
   Fig-23 JSON data format from the API server



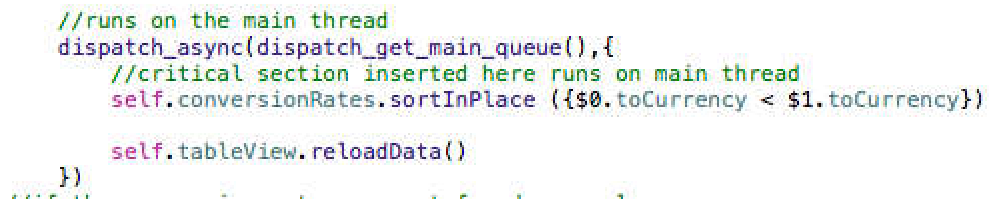
```
{
  "terms": "https://currencylayer.com/terms",
  "privacy": "https://currencylayer.com/privacy",
    "timestamp": 1430401802,
    "source": "USD",
    "quotes": {
        "USDAED": 3.672982,
        "USDAFN": 57.8936,
        "USDALL": 126.1652,
        "USDAMD": 475.306,
        "USDANG": 1.78952,
        "USDAOA": 109.216875,
        "USDARS": 8.901966,
        "USDAUD": 1.269072,
        "USDAWG": 1.792375,
        "USDAZN": 1.04945,
        "USDBAM": 1.757305,
    [...]
    }
}
```

4. Value for the “quotes” keyvalue is what we obtain.

Challenges Faced and Surmounted

Thread safe execution

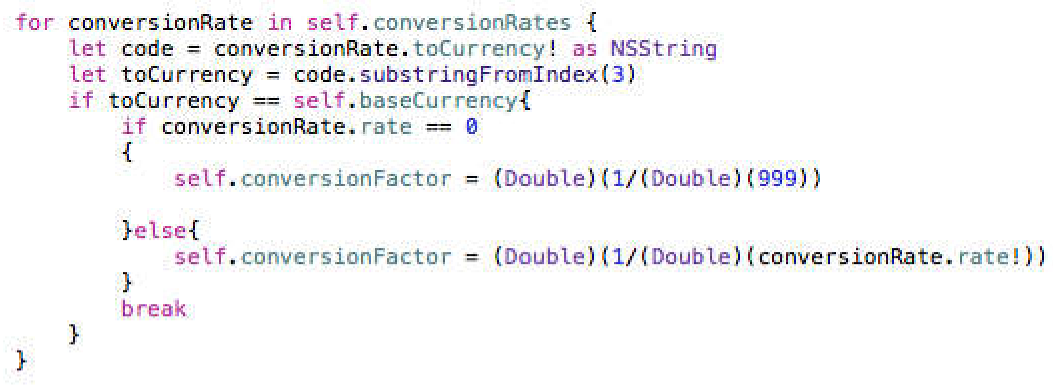
5. At a given time multiple threads are executed in the swift. Hence there should be accurate synchronisation between the data obtained from the API queries and the data that to be displayed on the tableview cells. To avoid the data hazard we used the following GCD function to make sure the critical section runs on the main thread.
6. Fig-24 despatch_asynch()



```
//runs on the main thread
dispatch_async(dispatch_get_main_queue(),{
    //critical section inserted here runs on main thread
    self.conversionRates.sortInPlace ({$0.toCurrency < $1.toCurrency})

    self.tableView.reloadData()
})
```

Computing Exchange Rates for the User Selected Base currency

7. The currency layer API gives the exchange rats only for USD as base currency. Thus to overcome it we adopted the technique of computed property. This enables the computation of exchange rates for any currency as base currency.
8. Fig-25 computing property for the exchange rate calculation



```
for conversionRate in self.conversionRates {
    let code = conversionRate.toCurrency! as NSString
    let toCurrency = code.substringFromIndex(3)
    if toCurrency == self.baseCurrency{
        if conversionRate.rate == 0
        {
            self.conversionFactor = (Double)(1/(Double)(999))

        }else{
            self.conversionFactor = (Double)(1/(Double)(conversionRate.rate!))
        }
        break
    }
}
```

Usage of Charts

9. This app required the depiction of data on the chart. But Swift did not have a framework for such visual representations. Thus we had to use an external API specific to this. This chart API has innumerous classes to be used. Linechart has been used in this project and has been customised for user appeasement

Parsing Query Data and Consideration of all Probable outcomes

10. The API queries may not necessarily result in expected results. Hence the various types of errors like invalid format of data, improper URL firing etc have been considered.

Image Formatting and writing to the resource bundle in the filesystem

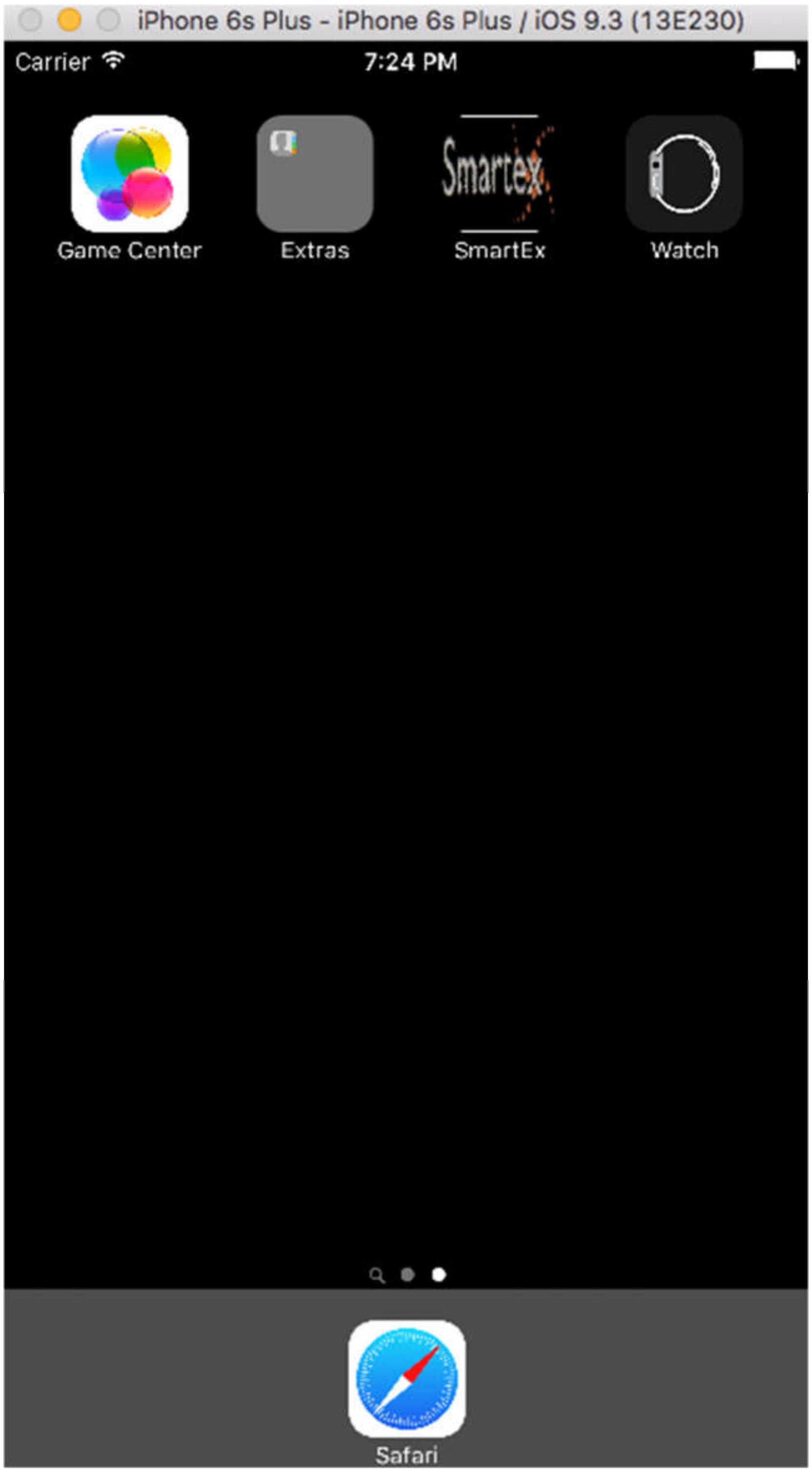
11. For the purpose of flag images they were renamed with 3 letter ISO currency codes and added to the assets.xcassets workspace

Following good design patterns

12. In order to enable future extensions of the project the necessary measures like enumerating the API query details, enumerating error types etc have been adopted. However, sensitive data such as API key etc have been access protected through the usage of private access identifier. Such measures ensures the Open to extension and closed to modification principle, Dependency Inversion Principle, Single responsibility Principle etc.

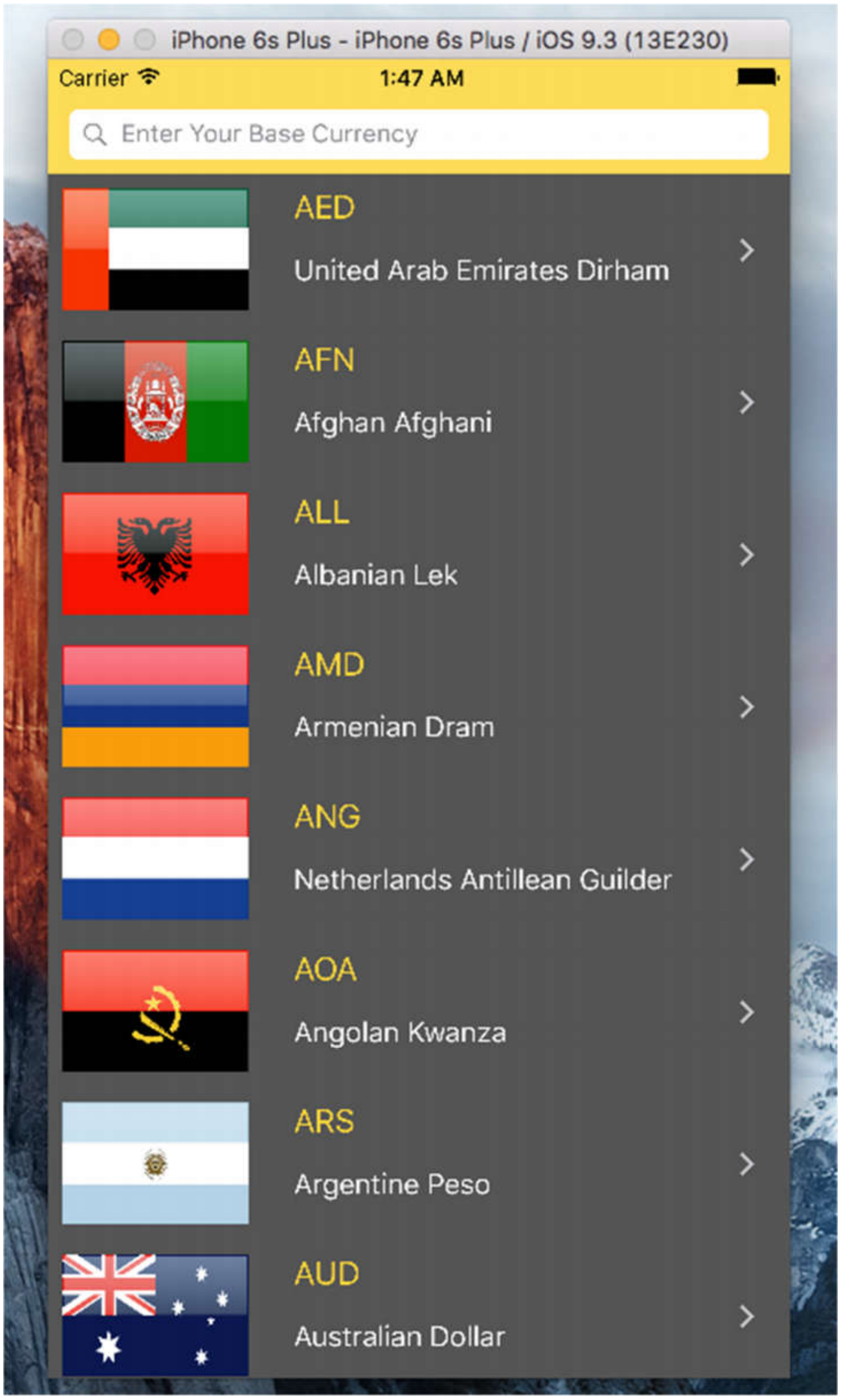
Screenshots

1. **Fig-26 AppIcon**



2. **<u>Fig-27 launch image</u>**

3. <u>**Fig- 28 homescreen**</u>



AED
United Arab Emirates Dirham

AFN
Afghan Afghani

ALL
Albanian Lek

AMD
Armenian Dram

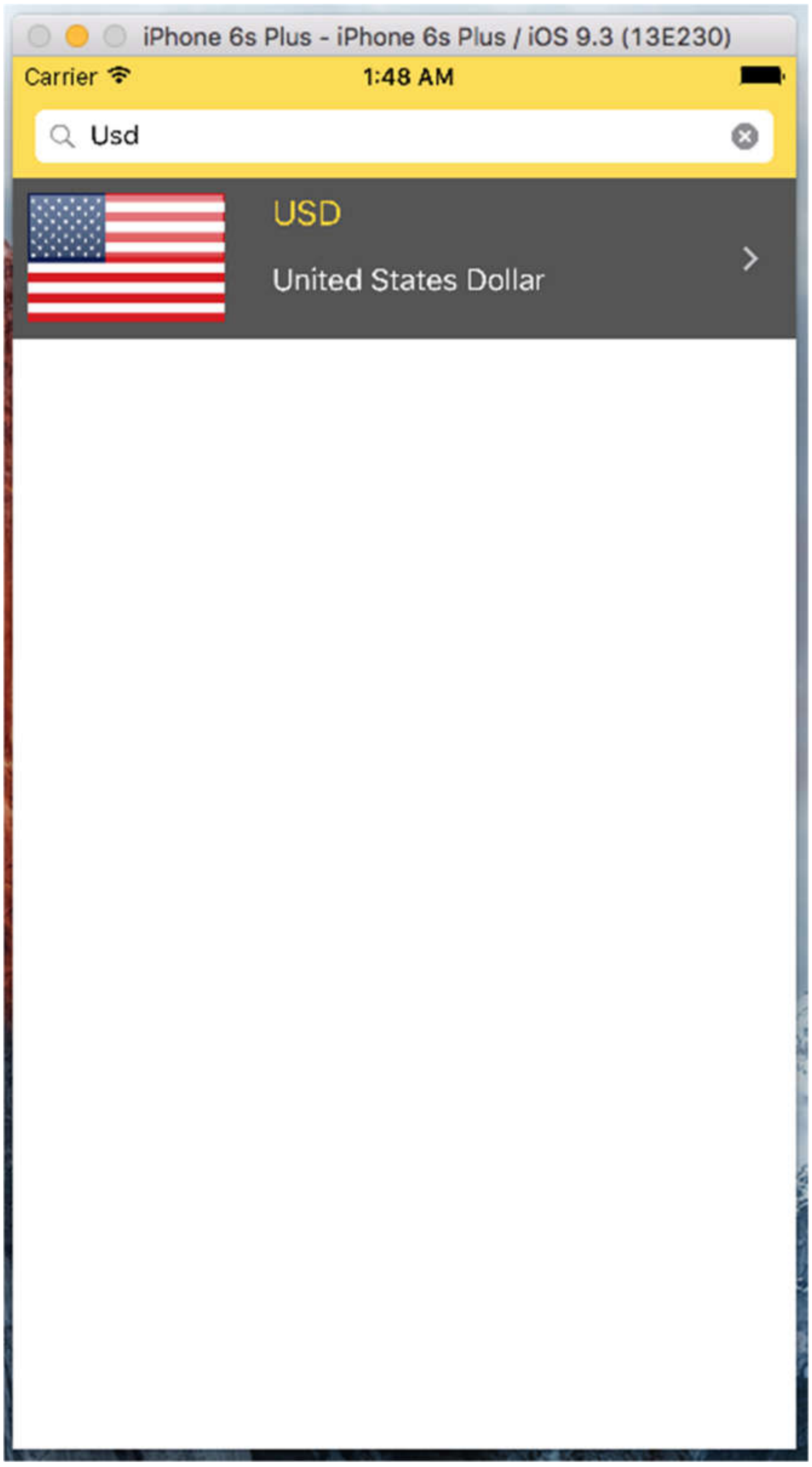
ANG
Netherlands Antillean Guilder

AOA
Angolan Kwanza

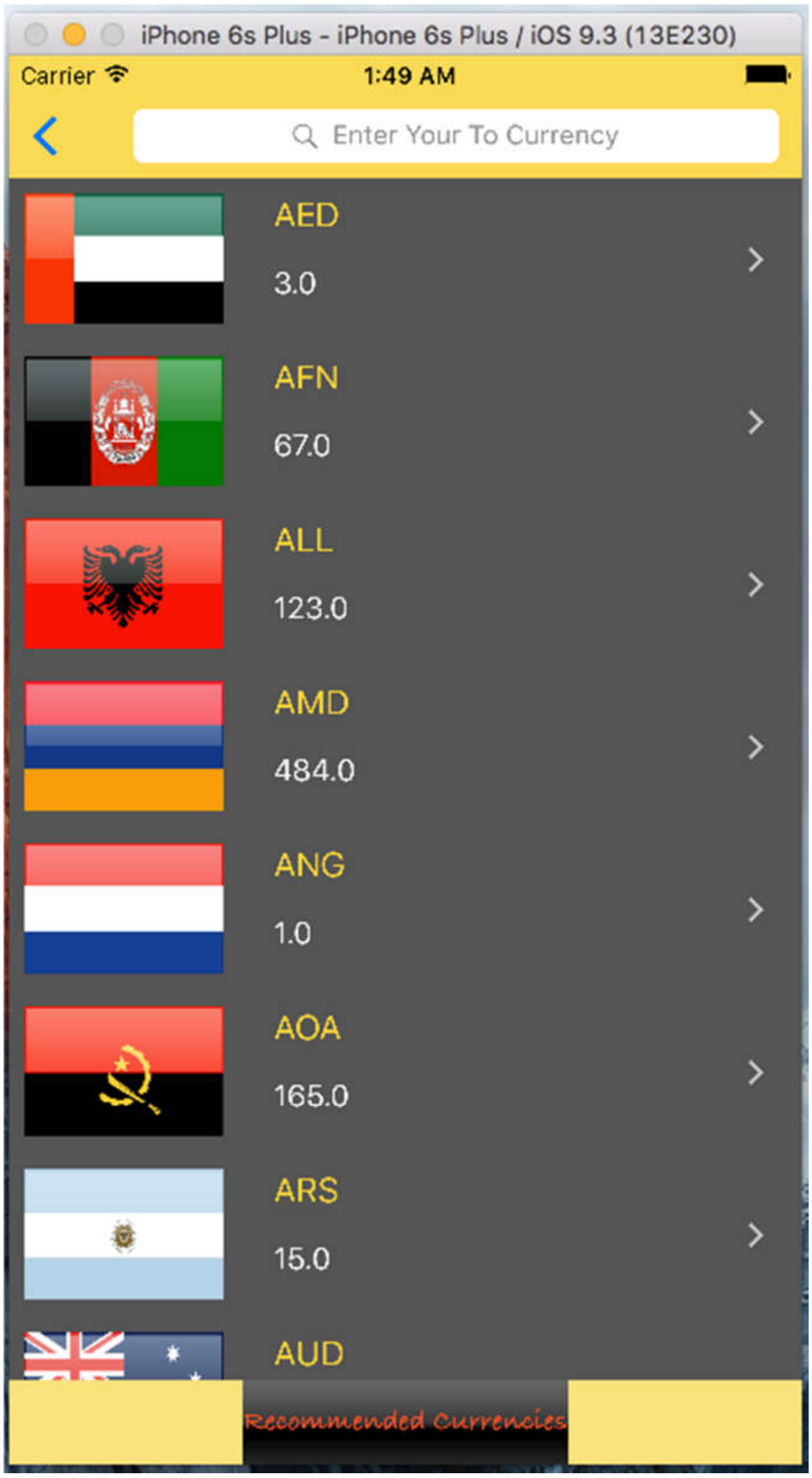
ARS
Argentine Peso

AUD
Australian Dollar

4. **<u>Fig- 29 base currency selection</u>**



5. **<u>Fig-30 to currency screen with live exchange rates with respect to the base currency</u>**



iPhone 6s Plus - iPhone 6s Plus / iOS 9.3 (13E230)

Carrier 1:49 AM

Enter Your To Currency

AED
3.0

AFN
67.0

ALL
123.0

AMD
484.0

ANG
1.0

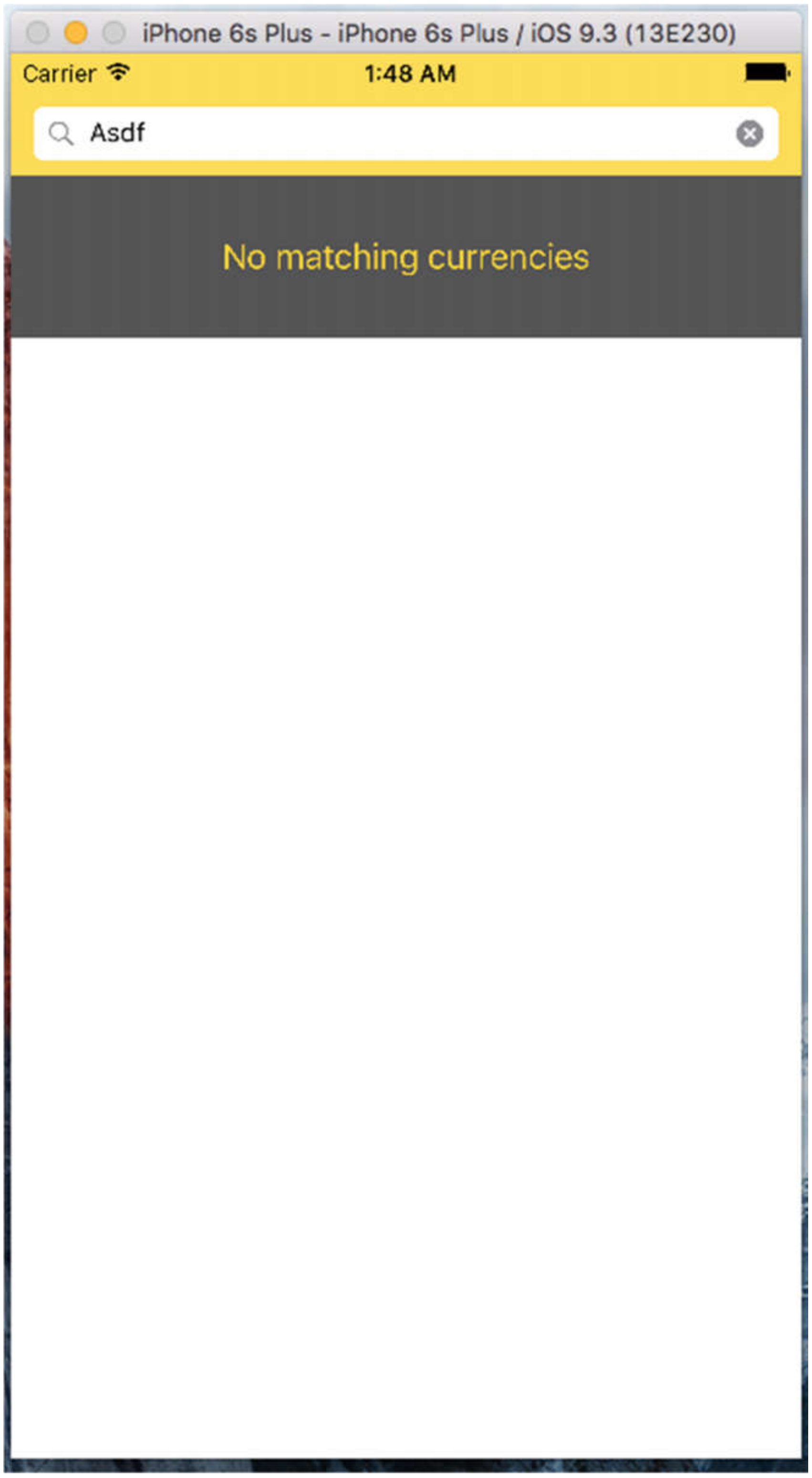
AOA
165.0

ARS
15.0

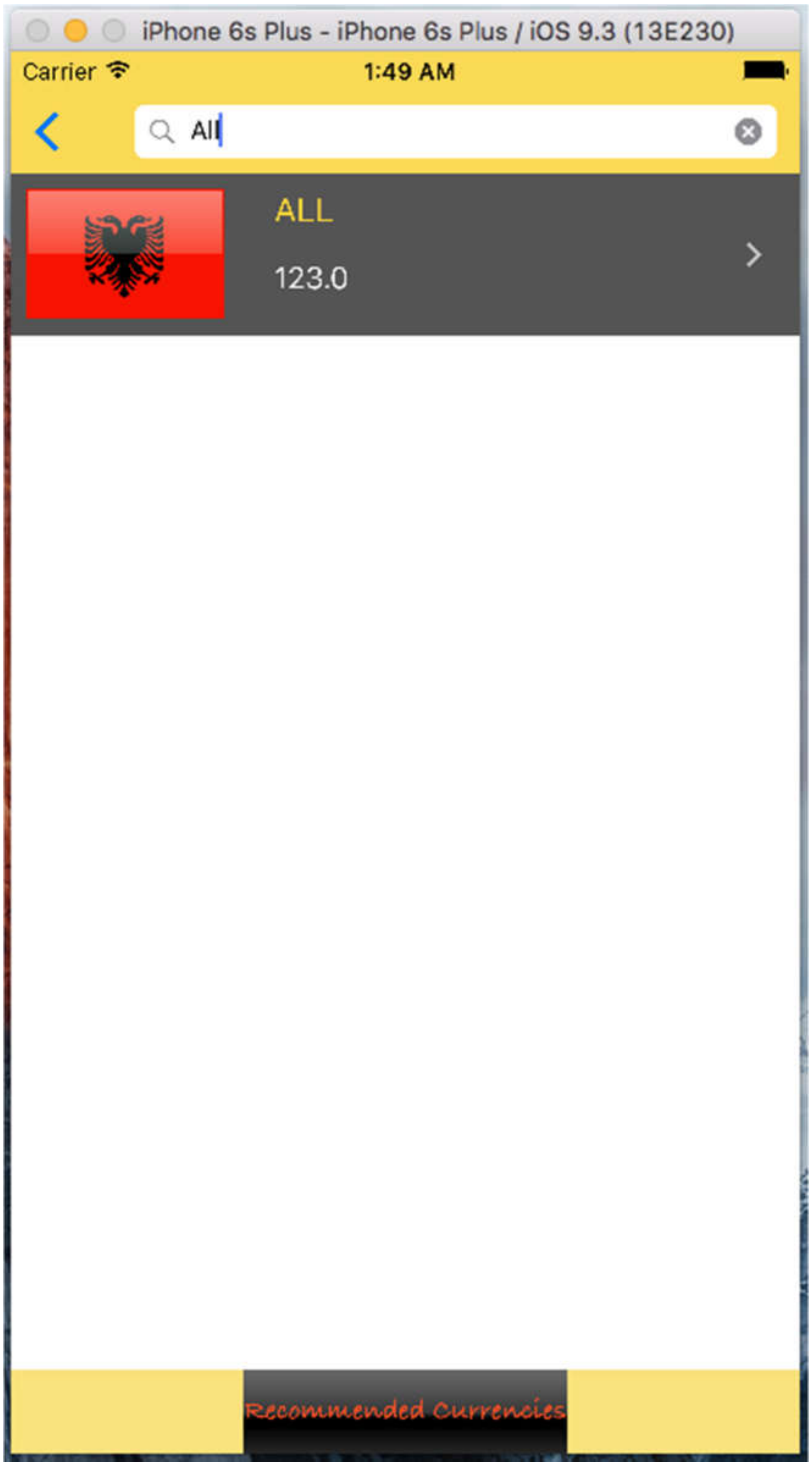
AUD

Recommended Currencies

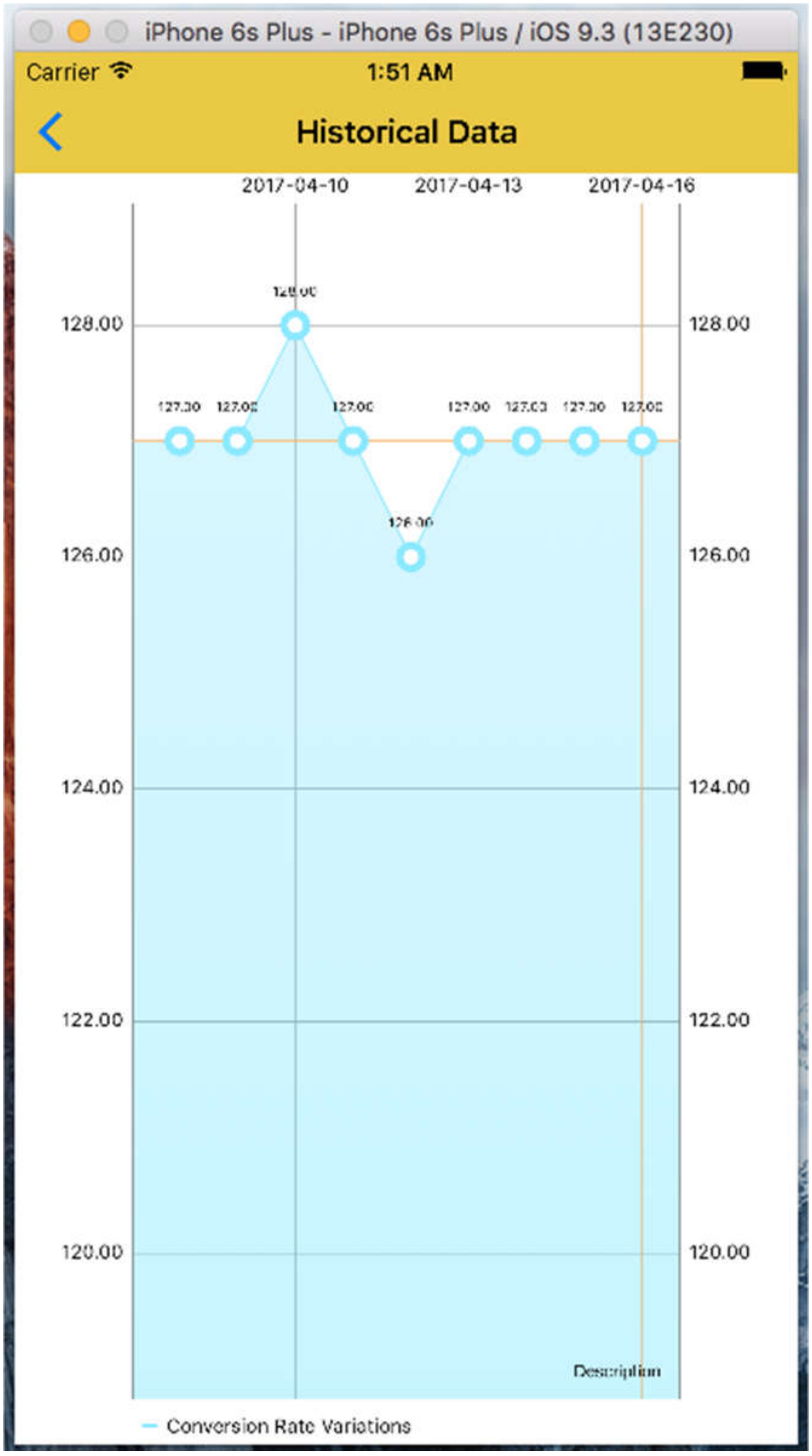
6. **<u>Fig- 31 no matching search</u>**



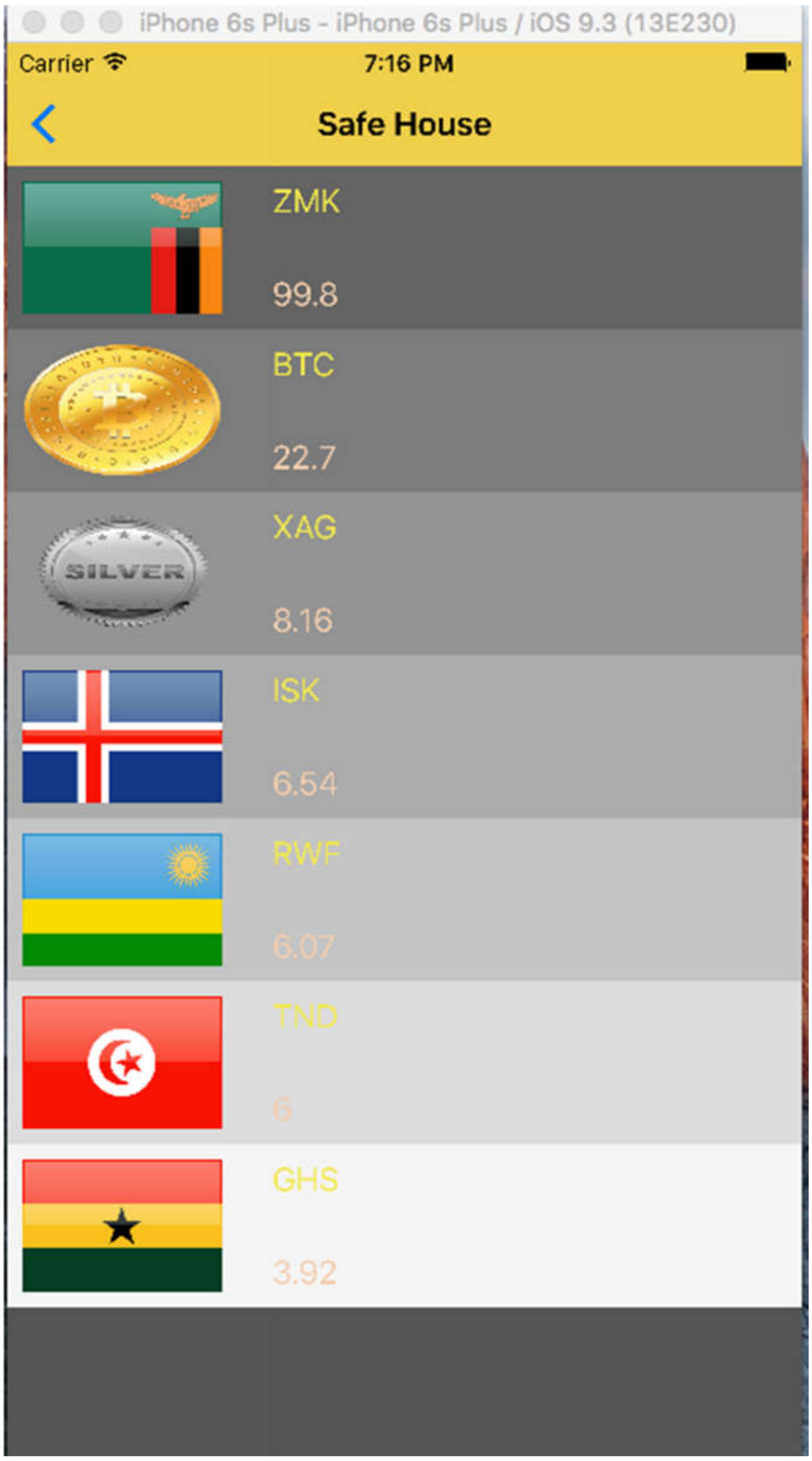
7. <u>**Fig- 32 to currency selection**</u>



8. <u>**Fig- 33 Historical data on chart**</u>



9. **<u>Fig- 34 recommended currencies which are risk less</u>**



iPhone 6s Plus - iPhone 6s Plus / iOS 9.3 (13E230)

Carrier 7:16 PM

Safe House

ZMK	99.8
BTC	22.7
XAG	8.16
ISK	6.54
RWF	6.07
TND	6
GHS	3.92

Summary and Future Work

Availability of user friendly, yet adept, apps are the need of time in the field of foreign exchange investment market. Our app tries to provide this through the adoption of simple interface. User will be relieved of knowing too many investment technical details and is handed over quantitatively decided safe currencies in which one can invest.

There are many possible extensions for the application. Improve the accuracy of recommendation system by using effective financial investment algorithms, providing text analytic inputs as part of fundamental analysis. Extending the app to provide linkage with the stock exchange and other financial institutions so that the user can be able to directly purchase using the app.

Team Members

Lakshmi Kanth Sandra : 229653990

Komal Rajendrakumar Gujarathi : 211778351