Main features

- Show the stock price graph of the past one month according to the given ticker
- Show the simulation result graph of the selected strategy and the stock price graph of the underlying stock on the screen
- Show the simulation result graph of the selected strategy and its parameters on the screen
- Display multiple simulation results in one graph. Clicking the curve to see its parameters
- Save the simulation result graph to camera roll
- Compute the value for American option

Important details

1. Stock Price & Option Value Query

Google finance API is used to get the stock price / option value. To use the API for stock price querying, first we shall replace TICKER, START and END in URL

```
"https://query.yahooapis.com/v1/public/yql?q=select Close from
yahoo.finance.historicaldata where symbol = '{TICKER}' and startDate = '{START}' and
endDate = '{END}' &env=store://datatables.org/alltableswithkeys&format=json"
```

with stock symbol, start date and end date respectively. Next we need to create a NSURLRequest `let request = NSURLRequest(URL:urlYql)` with default configuration:

```
let config = NSURLSessionConfiguration.defaultSessionConfiguration()
let session = NSURLSession(configuration: config)
```

Then we can start the request and get the results from the URL. The outcome is in JSON format, therefore a JSON object can be constructed by using the method `NSJSONSerialization.JSONObjectWithData`. The object is actually a dictionary, and we need to cast its type to NSDictionary in order to get the value.

To get option value by giving stock symbol and date, however, needs more work. The outcome of Google option chaining API is also in JSON format, that is, various key-value pairs. For instance, the key is “option type” and the value is “put” or “call”. BUT Google, in their infinite wisdom, doesn't enclose the key names in quotes, so the returned JSON is not well formed, which means it is not a valid JSON object. Therefore, the methods of NSJSONSerialization can’t be used to parse it. The solution is to add quotes for each key name with the help of regular expression:

```
optionChainString = optionChainString.stringByReplacingOccurrencesOfString("(\\w+)\\s*:",
    withString: "\"$1\":", options: NSStringCompareOptions.RegularExpressionSearch, range:
    optionChainString.startIndex ..< optionChainString.endIndex)
```

Then we can continue the JSON process like what we do in stock price querying and get the option price of a certain date.

2. Graph

I manage to add charts to the app using the ios-charts library by Daniel Cohen

Gindi. The library includes 8 chart types, whereas only line chart is used in the app. To use line chart, we need to create an empty view and set its class to LineChartView. Then use Ctrl+Drag to add an outlet of the view to the GraphViewController class. Since we need to draw the stock price graph as well as the simulation result graph, two views are needed as follows

```
@IBOutlet weak var stockPriceGraphView: LineChartView!
@IBOutlet weak var simulationResultGraphView: LineChartView!
```

Next we need to provide data for each graph. The line chart is based on two arrays, one for x-axis and the other for y-axis. In our app, x-axis is date (array of String) and y-axis is stock price / portfolio balance (array of Double). For a chart to display data, we need to create a LineChartData object and set it as the lineChartView's data attribute.

```
let months: [String]! = dates
let price = self.prices

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

let lineChartDataSet = LineChartDataSet(yVals: dataEntries, label: "Prices")
```

In the above code, we create an array of ChartDataEntry objects. The ChartDataEntry initializer takes the value of each data entry, the index of the entry the value corresponds to and an optional label. We then use this object to create a LineChartDataSet which is created by passing in the array of ChartDataEntry objects and a label to describe the data.

Finally, we use this to create a LineChartData object which we set as our chart view's data: `stockPriceGraphView!.data = lineChartData`.

Besides, we can customize the look of the chart view by changing some of its attributes. For example, when showing multiple simulation results in the same graph, we need to use different colors to represent different simulations. It can be done in this way `lineChartDataSet.setColor(graphColorArray[colorNumber%10])`, where graphColorArray is an array of UIColor.

Both stock price graph and simulation result graph are represented as line chart, supporting some gestures like dragging, panning and pinching. We don't even need to bother such things since the library implements all those gestures for us.

The library also helps to save the current state of a chart as an image. As in the app, we can save the graph to the camera roll. This feature is implements as follows

```
@IBAction func saveChart(sender: UIBarButtonItem) {
    let saveAlert = UIAlertController(title: "Save Image", message: "Save the image to camera roll",
        preferredStyle: UIAlertControllerStyle.Alert)

    saveAlert.addAction(UIAlertAction(title: "Cancel", style: .Cancel, handler: { (action: UIAlertAction!)
        in

    }))

    saveAlert.addAction(UIAlertAction(title: "Ok", style: .Default, handler: { (action: UIAlertAction!) in
        self.simulationDetailGraphView.saveToCameraRoll()
    }))

    presentViewController(saveAlert, animated: true, completion: nil)
}
```

The function above is an action of the button “save”. When user taps on the button, an alert dialogue will popover. If user taps on Ok, then method saveToCameraRoll will be called. It’s a method we get for free once we declare that our class implements ChartViewDelegate protocol.

```
class SimulationDetailViewController: UIViewController, ChartViewDelegate {
```

Moreover, when displaying multiple simulation results in one graph, user can tap on a curve and its parameters will appear below the graph. This feature is implemented by a method called chartValueSelected, which is also included in the ChartViewDelegate protocol.

```
func chartValueSelected(chartView: ChartViewBase, entry: ChartDataEntry,
    dataSetIndex: Int, highlight: ChartHighlight)
```

The last argument highlight represents the curve being selected, so we can identify the selected curve in this way

```
let selectedSimulation = simulationInfoArray[highlight.dataSetIndex]
```

3. Calendar

The calendar library is written in Objective-C, but I want to keep the project in pure Swift. So simply copying the library directory into our project is not an option. The only choice is to use CocoaPods. CocoaPods is a tool that manages library dependencies for the Xcode project. The dependency (the calendar library) for the project is specified in a single text file called a Podfile. CocoaPods will resolve dependencies between libraries, fetch the resulting source code, then link it together in an Xcode workspace to build our project. The benefit of CocoaPods is that we are able to create a more centralized ecosystem even if third party open-source libraries are used.

So first we need to create the Podfile and add our dependencies

```
platform :ios, '8.0'
use_frameworks!
target “PocketFinance” do
  pod 'THCalendarDatePicker','~>1.2.6'
end
```

Save the file to the project directory. Then change our current directory to the project directory and run $pod install. After that, a workspace will be created and from now on we shall use the workspace instead of the original xcode project.

The library itself is quite easy to use. First we need to declare that our class implements the `THDatePickerDelegate` protocol.

And the code below is all we need to initialize the calendar.

```
lazy var datePicker : THDatePickerViewController = {
    let picker = THDatePickerViewController.datePicker()
    picker.delegate = self
    picker.date = self.curDate
    picker.setAllowClearDate(false)
    picker.setClearAsToday(true)
    picker.setAutoCloseOnSelectDate(true)
    picker.setAllowSelectionOfSelectedDate(true)
    picker.setDisableYearSwitch(false)
    picker.setDisableFutureSelection(false)
    picker.autoCloseCancelDelay = 2.3
    picker.rounded = true
    picker.dateTitle = "Calendar"
    picker.selectedBackgroundColor = UIColor(red: 125.0/255.0, green: 208.0/255.0,
        blue: 0.0/255.0, alpha: 1.0)
    picker.currentDateColor = UIColor(red: 242.0/255.0, green: 121.0/255.0, blue:
        53.0/255.0, alpha: 1.0)
    picker.currentDateColorSelected = UIColor.yellowColor()
    return picker
}()
```

Future work

- At present I just assume that Google finance API always gives us correct data. When the network condition is bad, however, it may not return the desired result. So try to use the API asynchronously, which means the simulation will proceed only when the API fetches us what we want
- Implement those strategies using real time data apart from historical data
- Implement option pricing for European option
- The UI layout is only for an iPhone. It doesn't look as good on other iOS devices such as iPad or iPad mini. Some adaptations are needed to make the app look nice on a pad.