# Build Quick Start Challenge for R Tools 1.0 for Visual Studio 2017

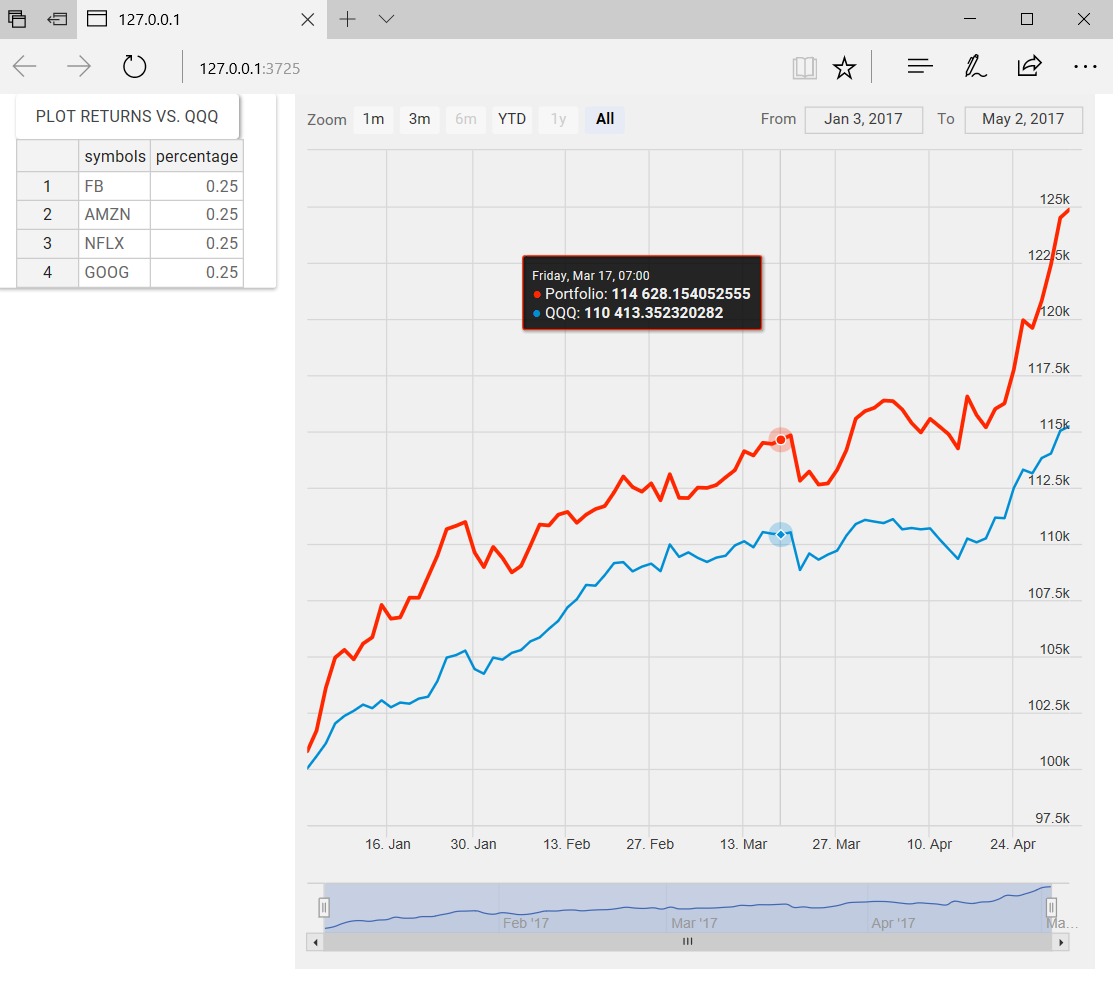
John Lam

## Your Challenge (If You Accept It!)

You will use RTVS to build a R Shiny application that lets you enter your own portfolio of stocks so that you can compare the performance of $100,000 invested in your portfolio vs. the NASDAQ 100 index that is tracked by the QQQ ETF. This is what the application looks like when you’re done. You can:

* Change the percentage of your investments (e.g, you can invest more in FB and less in NFLX)
* You can add/remove different stock ticker symbols
* You can interactively inspect the plot and zoom in on desired regions of interest

The entire application is built using R. No JavaScript or HTML required at all. Along the way, you will have a tour of some of the coolest features in RTVS in Visual Studio 2017. So let’s start!

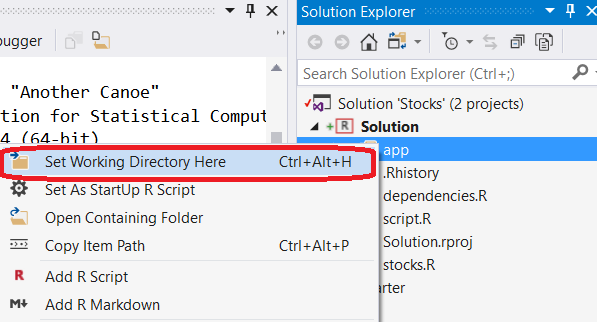


## Getting Started: Running the Solution

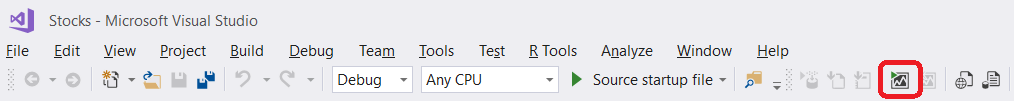
On your desktop, you’ll find a shortcut called Stocks. Double-click on that shortcut to open the lab solution in Visual Studio 2017. The solution is organized into two projects: **Solution** and **Starter**. The **Solution** contains the fully functional solution to the lab. If you get stuck, run out of time, or don’t believe the app actually works, you can always just run the project in the **Solution** directory.

Here is how you run a Shiny app:

1. Set the working directory to the **app** subdirectory of the **Solution** project by right-clicking on the **app** icon in the solution, and running the **Set Working Directory Here** command.



1. Click on the Run Shiny App toolbar button to start the stocks app. You should see the default web browser launch with the app up and running.



Next, try clicking on the PLOT RETURNS VS. QQQ button. You should see a plot that resembles the image at the top of this lab manual. Try editing some of the values, or changing the names of some of the stock symbols. Note that error handling is an exercise left to the reader!

You can also add additional columns to the grid by right clicking on the grid and inserting additional rows above or below the cursor. Try creating a new row, or removing a row. You’ll need to click on the PLOT RETURNS VS. QQQ button to see changes in the plot. See if you can beat the QQQ index with your own portfolio.

## Building your own app

The Stocks app uses several different R packages. These dependencies are all gathered together for you in the **dependencies.R** file. Open the file up in Visual Studio by double-clicking on it. You’ll see that we depend on four different packages:

* [shiny](http://shiny.rstudio.com/gallery/): package for building web applications in R
* [quantmod](http://www.quantmod.com/): package for working with financial time series data
* [highcharter](http://jkunst.com/highcharter/showcase.html): package for high quality HTML/CSS/JavaScript plots
* [rhandsontable](http://jrowen.github.io/rhandsontable/): package for editable, interactive HTML grid

The code in the **dependencies.R** file uses a helper function **install\_if\_not\_present** to check to see if a package was installed already before attempting to install it.

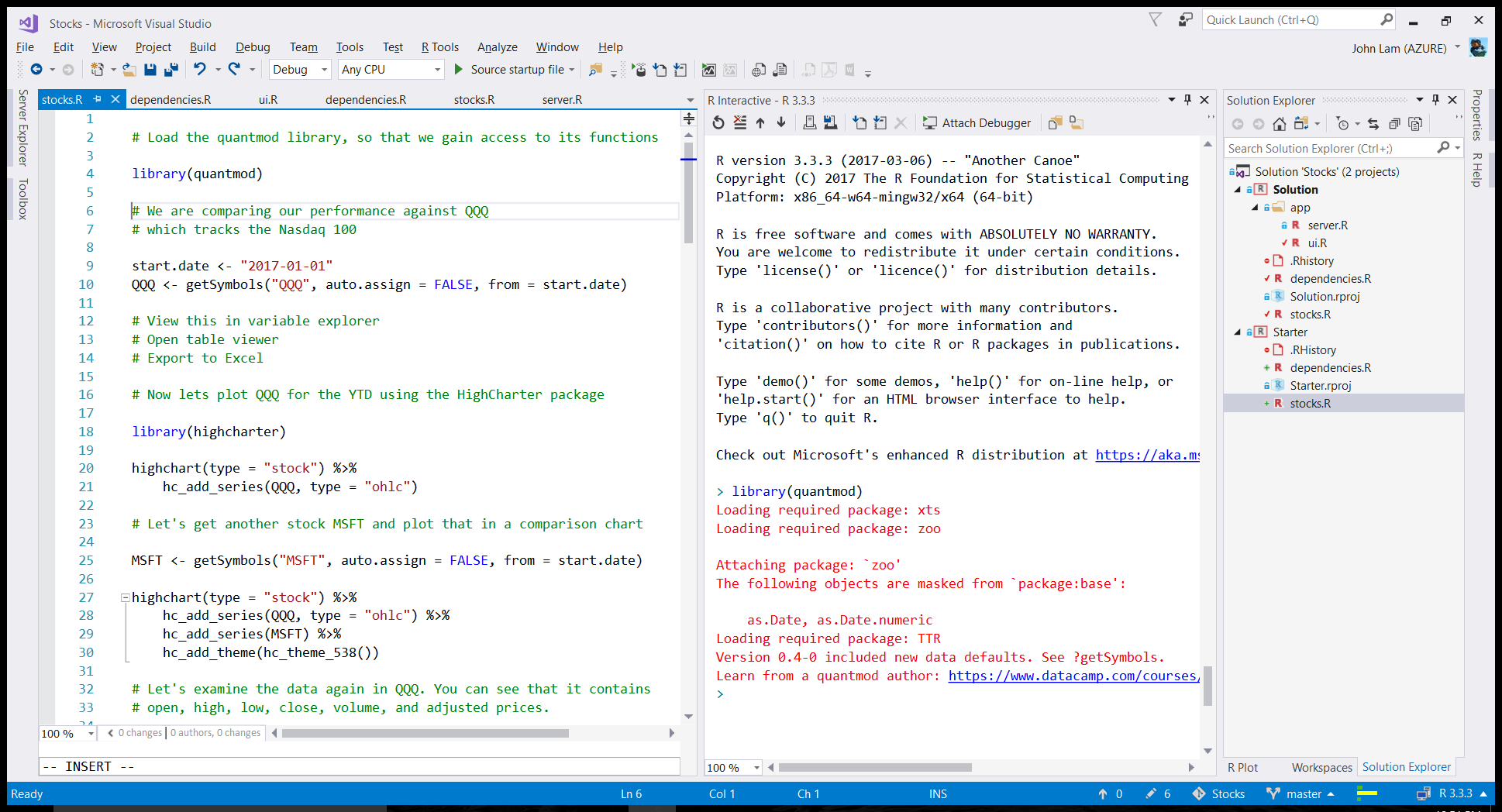
Right click on the file in Solution Explorer and **source** it. The source command instructs the R interpreter to execute the contents of the file.

## Interactive Execution

Open the **stocks.R** file where we’ll learn about the R programming language while interactively executing the code in the file. Below is a picture of the way that I like to arrange my windows on a 1920 x 1080 monitor. I have, on the left hand side, the **stocks.R** file opened in the editor. To the right of it, I have the R Interactive Window (sometimes called the REPL – Read-Eval-Print-Loop window) open.

I find this arrangement of windows especially useful for interactively executing code. While writing the lab, I would write the code that I want to write in the editor window on the left, and send the lines of code that I wanted to run to the REPL window on the right. There are two different ways that you can interactively execute some code in the REPL:

1. Select a block of text in the editor and press CTRL+ENTER to send the selection to the REPL window and executed.
2. Press CTRL+ENTER without a selection, and the current line under the caret will be sent to the REPL window and executed.



Place your cursor anywhere on the line that contains the **library(quantmod)** command and press CTRL+ENTER. You should see that command copied to the REPL window and run, producing the red lines of output that you see in the image above.

Follow along with the explanations in the **script.R** file to work through an example of how to download stock data from the Internet using functions in the **quantmod** library, and writing code to compute returns on a hypothetical $100,000 investment in a portfolio of stocks. You will get to see another key library at work as well, **highcharter**, which is a library that produces very high-quality charts in your default web browser.

If you’re impatient and just want to see what happens you could either:

1. Source the **stocks.R** file, which will run all of the code (and produce 4 plots)
2. Select all the lines of code in the file via CTRL+A, and then hitting CTRL+ENTER to execute all of those lines in the REPL window.

However, I strongly advise that you run the code line-by-line and read the comments to get the most out of your experience.

## Building the Shiny Application

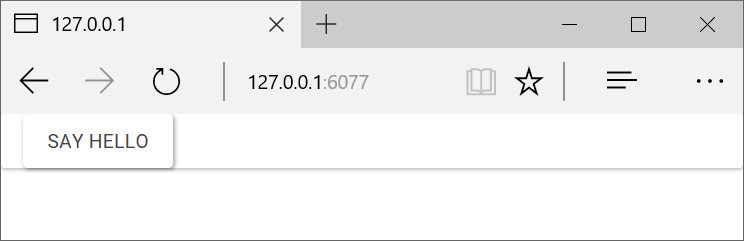
Shiny is a web application framework for R. You write R code, and Shiny takes care of generating the HTML, CSS and Javascript for your application. Shiny is a *reactive* framework, which requires understanding of the reactive programming model. Here’s a [great overview](https://shiny.rstudio.com/articles/reactivity-overview.html). For the purposes of this lab, we’ll be looking at a small subset of what Shiny has to offer. But it will give you enough to get started, and provide you with a solid platform for further experimentation.

Begin by running the Shiny application in the **app** directory of the **Starter** project. Just like you did in the solution, make sure that you right-click on the **app** directory under Starter and run the **Set Working Directory Here** command to make sure that we run the correct version of the app.

Note: If the Shiny solution is still running in RTVS at this point, make sure you stop execution first. You can do this by clicking on the **Stop Shiny App** toolbar button in Visual Studio.

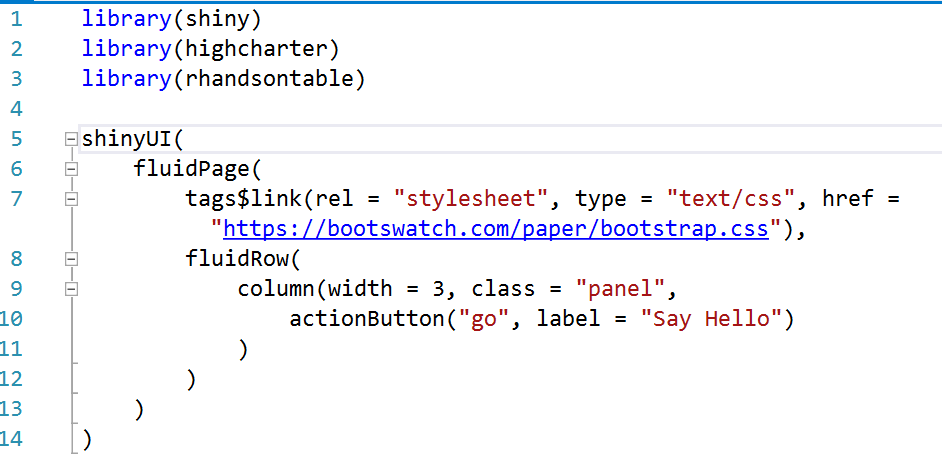


When you run the Starter app, you’ll see the default browser open, with a web page that has a button that you can click on:

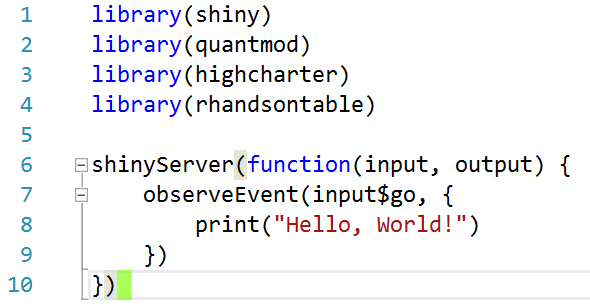


When you click on the button, you should see Hello, World show up in the Interactive Window.

Your Shiny application lives in two different files. The first, **ui.R** defines the layout of the UI elements of your app. Below, you’ll see that there is a function, **actionButton**, that creates the “Say Hello” button (id == go) that you see in the page.



The code that is invoked in response to the click lives in **server.R**. The **observeEvent**() function registers a function that is invoked in response to the user clicking on the button with the **id** == go.



Let’s begin by modifying **ui.R** to include UI elements necessary for our application. We’ll need placeholders for:

1. The grid (supplied by the [rHandsontable library](https://jrowen.github.io/rhandsontable/)) that contains the portfolio of stocks
2. The chart (supplied by the [highcharter library](http://jkunst.com/highcharter/)) that contains the comparison plot

Replace the **fluidRow()** function call in **ui.R** with this code:

fluidRow(

column(width = 3, class = "panel",

actionButton("go", label = "Plot returns vs. QQQ"),

rHandsontableOutput("hot")

),

column(width = 9,

highchartOutput("chart", height = "700px")

)

)

This code also contains a new definition for our **go** actionButton where its caption now reads “Plot returns vs. QQQ”.

In the **server.R** file, we’ll need to write some additional code that:

1. Computes the market value of the stocks in a portfolio.
2. Generates a highcharter plot given a portfolio object
3. Initializes the data grid with initial values of the portfolio (FANG at 25% even weight).

First, let’s add the code that computes the daily market value of the portfolio, given a portfolio object and the total initial investment in that portfolio. The code below should be really familiar to you, as it’s exactly the same code that you saw in the first part of the lab, but with the explanatory comments elided as you now fully understand what it does 😊

starting.investment = 100000

start.date = "2017-01-01"

compute.portfolio.daily.market.value <- function(portfolio, dollars) {

portfolio <- cbind(

portfolio,

dollars = portfolio$percentage \* dollars)

df <- NULL

compute.daily.market.value <- function(symbol, dollars) {

symbol.data <- getSymbols(

symbol,

auto.assign = FALSE,

from = start.date)

shares <- dollars / as.numeric(first(Op(symbol.data)))

market.value <- Ad(symbol.data) \* shares

if (is.null(df)) {

df <<- data.frame(market.value)

}

else {

df <<- cbind(df, data.frame(market.value))

}

}

mapply(

compute.daily.market.value,

portfolio$symbols,

portfolio$dollars)

return(cbind(df, data.frame(Total = rowSums(df))))

}

It is worth asking yourself the following questions, to test your understanding from the first part of the lab:

1. What does the [mapply](https://stat.ethz.ch/R-manual/R-devel/library/base/html/mapply.html)() function do? How is it an example of the general R principle of “if you’re writing a loop, you’re probably doing it wrong”?
2. What does the <<- operator do?
3. How does the [cbind](https://stat.ethz.ch/R-manual/R-devel/library/base/html/cbind.html)() function help us construct the result of our computation? What does the result look like?

Next, let’s add the code that responds to the UI events. Copy and paste the code below and replace the existing **shinyServer**() function definition in the **server.R** file.

shinyServer(function(input, output) {

v <- reactiveValues(portfolio = NULL, index = NULL)

observeEvent(input$go, {

if (is.null(input$hot)) return()

# Compute the value of the portfolio and the comparison

# portfolio (we do QQQ here)

portfolio = hot\_to\_r(input$hot)

v$portfolio <- compute.portfolio.daily.market.value(

portfolio,

starting.investment)

v$index <- compute.portfolio.daily.market.value(

data.frame(

symbols = c("QQQ"),

percentage = c(1.0),

stringsAsFactors = FALSE

),

starting.investment

)

})

convert.totals.dataframe.to.xts <- function(df) {

return(xts(df$Total, as.POSIXct(rownames(df))))

}

output$chart <- renderHighchart({

if (is.null(v$portfolio) || is.null(v$index)) return()

highchart(type = "stock") %>%

hc\_add\_series(

data = convert.totals.dataframe.to.xts(v$portfolio),

name = "Portfolio") %>%

hc\_add\_series(

data = convert.totals.dataframe.to.xts(v$index),

name = "QQQ") %>%

hc\_add\_theme(hc\_theme\_538())

})

output$hot <- renderRHandsontable({

if (!is.null(input$hot)) {

portfolio = hot\_to\_r(input$hot)

} else {

portfolio = data.frame(

symbols = c("FB", "AMZN", "NFLX", "GOOG"),

percentage = c(0.25, 0.25, 0.25, 0.25),

stringsAsFactors = FALSE)

}

rhandsontable(portfolio) %>%

hot\_table(highlightCol = TRUE, highlightRow = TRUE)

})

})

Let’s walk through what this code does:

1. There is an anonymous function wrapper that is the input/output loop of the Shiny application. The **input** and **output** parameters contain references to the input and output objects of the Shiny app.
2. Within this function, we write our code that handles events. At the top level scope, we call the **reactiveValues**() API to define two new session variables for our application: **portfolio** and **index.** Further down in our code, we’ll initialize those variables.
3. You can see the call to **observeEvent**() that contains a function that is invoked in response to the user clicking on the “Plot returns vs. QQQ” button. When the user clicks on this button, we’ll compute the daily market values of the stocks in our portfolio, as well as compute the daily market values of a *second portfolio that only contains the QQQ stock*. Note that we don’t do any plotting here – we are merely computing the values of our portfolio and assigning them to our **portfolio** and **index** session variables.
4. The **renderHighChart**() function defines a function that is invoked to render the chart. It will only render if it sees that the **portfolio** and **index** session variables are assigned a value other than their default value of **NULL**. If it sees that, it will use the familiar code that we saw in the first part of the lab that generates a plot of two series containing the daily totals from each one of our two portfolios: **portfolio** and **index**.
5. The **renderRHandsontable**() function defines a function that creates a dataframe that contains our starting portfolio of FANG stocks with equal weight distribution. This is what happens the first time the function is called. On subsequent calls to this function, we’ll see that we simply take what was passed to us and re-render it as output. This is a common thing that we see in HTML as each time a callback happens, it results in generation of HTML required to render the result on the client.

Once again, when looking at the code, it’s worth asking yourself some questions to test your understanding of what you’ve learned. You’ll need to spend some time reading the [Reactivity Overview](https://shiny.rstudio.com/articles/reactivity-overview.html)document referenced earlier.

1. What is the role of the [observeEvent](https://shiny.rstudio.com/reference/shiny/latest/observeEvent.html)() function?
2. Why are we separating the inputs and the outputs?
3. Why do we define session variables (variables whose lifetime equals that of the application) using the [reactiveValues](https://shiny.rstudio.com/reference/shiny/latest/reactiveValues.html)() API vs. just defining them as regular variables?