Below is the complete code for my first shiny app. Only 56 lines (a good chunk of which are comments and styling) in hopefully readable formatting. I considered it fitting to base it on the classic coin flip experiment which results in either Heads or Tails:

# preparations; required libraries

library(shiny)

library(dplyr)

library(tibble)

library(stringr)

library(ggplot2)

# user interface elements and layout

ui <- fluidPage(

titlePanel("Heads or Tails"),

sidebarLayout(

sidebarPanel(

sliderInput(inputId = "n", label = "Number of flips:",

min = 10, max = 1000, value = 500),

sliderInput(inputId = "prob", label = "Success rate:",

min = 0, max = 1, value = 0.5),

tags$div(tags$p(HTML("

Find the annotated code")),

tags$a(href=post, "in this blog post."))

),

mainPanel(plotOutput(outputId = "bars"))

)

)

# server-side computations

server <- function(input, output) {

# the bar plot

output$bars <- renderPlot({

# most of this is for ggplot2; note the input$x syntax

flips <- tibble(flips = rbinom(input$n, 1, input$prob)) %>%

mutate(flips = if\_else(flips == 1, "Heads", "Tails"))

flips %>%

count(flips) %>%

ggplot(aes(flips, n, fill = flips)) +

geom\_col() +

geom\_label(aes(flips, n, label = n), size = 5) +

theme(legend.position = "none",

axis.text = element\_text(size = 15)) +

labs(x = "", y = "") +

ggtitle(str\_c("Results of ", input$n,

" flips with Heads probability ",

sprintf("%.2f", input$prob)))

})

}

# run it all

shinyApp(ui = ui, server = server)

All you need to do at this stage is to (have the required libraries installed and) copy/paste the code above into an active R session. Try it out!

The app allows you to choose the number of coin flips as well as the probability for Heads using slider bars. It visualises the resulting total numbers of Heads vs Tails as a reactive bar plot. Given the functionality of this app, 56 lines is not too bad, is it? Let’s dissect the code element by element!

**Preparations:** Before we get to the interesting parts, the first five lines define and load the packages the script needs. This is unrelated to shiny (other than loading it):

library(shiny)

Note, that shiny web apps on shinyapps.io apparently need explicit library calls and that my normal approach of using invisible(lapply()) led to some confusing errors before I figured it out. Besides the libraries, I’m also including the url for this post as part of the preparation.

**The shiny code** is structured into two main elements: (i) a **user interface (UI)** definition and layout, and (ii) the **server-side computations** producing the data for plots (or tables, or other output elements). At the end, there is always a **call to the shinyApp function** which renders the whole thing.

The **UI setup** starts with:

ui <- fluidPage(

which defines the internal name of the UI as ui (*very surprising; I know*). The fluidPage environment creates an output html that automatically adjusts to the size and shape of your viewer window. This seems to be the layout you would choose most often. The 2 alternatives are a fixedPage or a navbarPage which gives you a top-level navigation bar.

Inside our fluidPage we have the UI elements. The first one gives your app a title:

titlePanel("Heads or Tails")

Nothing too complex here. The next element is the sidebarLayout; as in “a layout that contains a sidebar” (as opposed to “a layout for the sidebar only”).

sidebarLayout(

sidebarPanel(...),

mainPanel(...)

)

This layout has always two elements: the sidebarPanel and the mainPanel. You can browse other layout options here, including grids and tabs.

The sidebarPanel typically contains the control widgets. Those widgets are what the users interact with.

Here, we are using a sliderInput to allow the user to select the number of coin flips (in a range from 10 – 1000) and the probability for Heads (in a range from 0 – 1):

sidebarPanel(

sliderInput(inputId = "n", label = "Number of flips:",

min = 10, max = 1000, value = 500),

sliderInput(inputId = "prob", label = "Success rate:",

min = 0, max = 1, value = 0.5),

)

Both sliders have the same syntax:

* min, max, and value define the slider range and the default value at which the slider sits upon loading the app. Those parameters are specific to the slider widget.
* label is the text explaining to the user what the slider is being used for.
* the inputID is important, since it will be used in the server-side part of the app to assign inputs to outputs. Note, that we call the number of flips n and the probability for Heads prob.

Other available widgets include checkboxes, radio buttons, or text input; each with their own specific parameters besides InputID and label.

Ǹote, that besides widgets and plots, html content or formatting can be added inside a Panel method. In the code I’m inserting a short paragraph and the hyperlink to this blog post:

tags$div(tags$p(HTML("

Find the annotated code")),

tags$a(href=post, "in this blog post."))

The mainPanel typically contains the rendered reactive output. This object will change immediately when the user selects a different input (here via the sliders). We choose a plot because plots are awesome:

mainPanel(plotOutput(outputId = "bars"))

* similar to the inputID above, the outputID connects UI elements to server computations.
* besides the plotOutput function, there are other functions to produce tables, images, text, and more.

Now the 2nd part: the **server setup**. Here is where all the computations happen that produce the data for our output elements based on the input parameters. This part is close to a typical R workflow, in that you build your plots or tables to communicate insights. The only difference is that parameters are passed from the input UI, and that none of the possible parameters should break your plots.

In the code, the server function builds a list-like object output based on the user input:

server <- function(input, output) {

output$bars <- renderPlot({})

}

* We define a single output: a plot via renderPlot. Other render function include renderImage or renderTable. You can add as many output elements as you need.
* The plot is assigned to output$bars. This means that it becomes an element in the output list (the only element in our case). The name bars needs to match the outputId = "bars" in our UI mainPanel.

Now, the code inside renderPlot is re-run every time the user changes the input parameters. In our example, I used some ggplot2 styling to make the plot look nicer. Here is an alternative one-liner using only base R, to emphasise the shiny elements. Go on and replace the renderPlot call in the starter code with this one to see what happens:

output$bars <- renderPlot({

barplot(table( rbinom(input$n, 1, input$prob) ))

})

* In both versions, the rbinom function does all the work by creating a list of n random numbers following a binomial distribution with a success probability of prob.
* Note, how the two input parameters are being passed as elements of the input object. Their names, n and prob need to match the respective inputIds in the UI part.

Finally, don’t forget the line that **runs the whole thing:**

shinyApp(ui = ui, server = server)

And that’s it! This is the main technical concept. The rest is the creative part: figuring out what to display with which user inputs. (Well, there’s also loading datasets and R scripts as well as streamlining bulky apps.)

Except, we’re not quite done yet. Copy/pasting code into the R console is *not quite* the best way to showcase your app. Here’s how to do it properly:

* Paste all the starter code above into a single file called app.R.
* Put that file into a sub-directory of your choice (e.g. ./headsortails/).
* And call shiny::runApp("headsortails") from an R session running in the parent directory of that subdirectory.

Each app.R should live in its own sub-directory. They are called via the names of their sub-directories. (Note, that the convenience of having both UI and server in the same file was not always possible. Old shiny versions required two separate ui.R and server.R files; a structure that’s still supported).

Finally, shiny apps are ideal to be shared online since they are reactive HTML. You can run your own shiny server to do this, especially if you have many different apps to showcase. For your first steps, I recommend using shinyapps.io, run by the omipresent Rstudio folks. They have a free tier allowing you to host 5 apps running for a maximum 25 hours per month. That’s plenty of resources to get your feet wet.

As indicated at the beginning, I’m using shinyapps.io to host the version of the app that is included above. However, you cannot embed shiny elements directly into a blogdown post like this one, since those posts are static. Above, I used the little trick of embedding the link to my shiny app via the HTML iframe tag. Like this: