Shiny Apps in R

Matt Hewitt

2024-08-21

## Shiny Structure

Shiny apps will all have the same 3 parts, some may have MUCH more, but all will have these three basic parts.

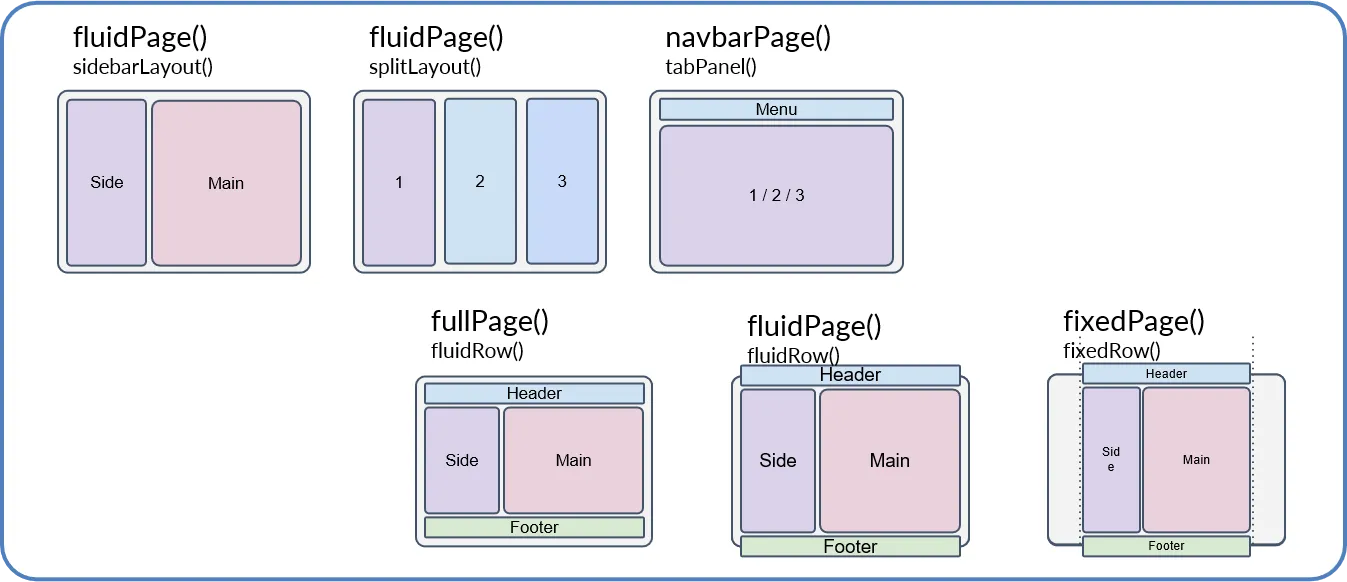
### Global

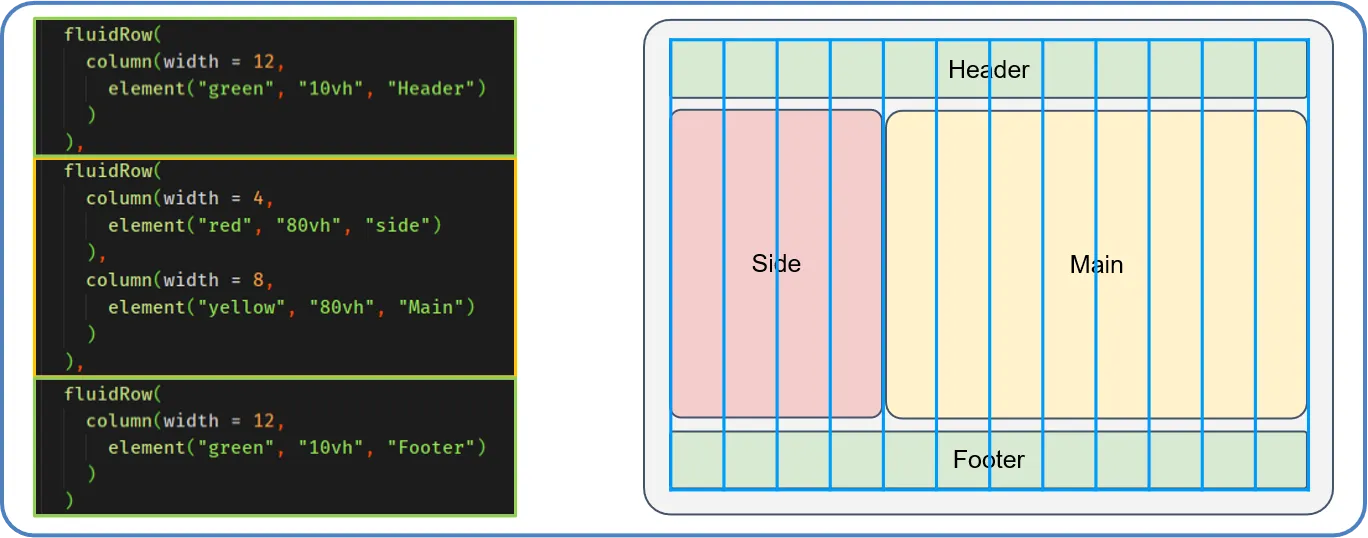
This part of the app is intended for you to ‘get things set up’. This is where you would load all the necessary packages, load data, connect to databases, source other files/scripts, any any other pieces of information your app needs to operate. This is usually not needed for simple apps, but can be extensive in larger ones.

### UI (User Interface)

This is the portion of the app that the user actually sees and interacts with. It can be as specific or as general as needed, usually very specific for basic apps and general for more complicated ones. If you do not need things in the UI to change while the app is being used, all UI structure can be built in the UI portion of the code. However; if the presence of things, such as drop down menus or tables, is dependent on inputs from the user, then that portion of the UI will end up being built in the Server portion of the code.

There are many packages that provide pre-built layouts, my favorite is shiny dashboard. Think of it as base R graphics and ggplot2, the base graphics work, but ggplot tends to be a lot prettier and more flexible. Shiny dashboard has some great resources for helping you build intuitive and aesthetically pleasing UI’s (<https://rstudio.github.io/shinydashboard/>).

There is a structured way that UI’s are built. We will use the *sidebarLayout()* layout further down in this document. Here are some examples of shiny’s default layouts:  


Within the sections of these layouts (or just a blank page, if you choose to go that route) there is a structure for placing items on the page. Items include anything, user inputs, outputs, tables, figures, boxes, titles, literally anything you want to place on the page. The method for ‘mapping’ where items are placed follow this structure:  


### Server

This is where the magic happens! This is where all processes and operations live that dictate the behavior of your app. The app will continuously be observing the server file and will run its different portions given different user inputs.

The App will run these pieces in a particular order. It will start with the **Global** portion, then the **UI**, and lastly the **Server**. The **Global** and **UI** will only be run once, when the app start up, but the server will constantly be observed and portions will be re-run given inputs from the user.

Lets look at a basic app. The following is the default example present in every new shiny file:

## Global -------------------------------------------------- Load packages  
library(shiny)  
  
 ## UI ------------------------------------------------------ Create User Interface layout  
ui <- fluidPage(  
  
 # Application title  
 titlePanel("Old Faithful Geyser Data"),  
  
 # Sidebar with a slider input for number of bins   
 sidebarLayout(  
 sidebarPanel(  
 sliderInput("bins",  
 "Number of bins:",  
 min = 1,  
 max = 50,  
 value = 30)  
 ),  
  
 # Show a plot of the generated distribution  
 mainPanel(  
 plotOutput("distPlot")  
 )  
 )  
)  
  
 ## Server -------------------------------------------------- Define Server logic  
server <- function(input, output) {  
  
 output$distPlot <- renderPlot({  
 # generate bins based on input$bins from ui.R  
 x <- faithful[, 2]  
 bins <- seq(min(x), max(x), length.out = input$bins + 1)  
  
 # draw the histogram with the specified number of bins  
 hist(x, breaks = bins, col = 'darkgray', border = 'white',  
 xlab = 'Waiting time to next eruption (in mins)',  
 main = 'Histogram of waiting times')  
 })  
}  
  
 ## --------------------------------------------------------- Run the application   
shinyApp(ui = ui, server = server)

In this app you can see the 3 basic parts. This particular app is a ‘one file’ app. Meaning all portions of the application are housed in one file. Alternatively, you can split all the portions of this up into different files. Meaning, your **Global**, **UI** and **Server** portions will all be in their own .R file.

But, we will continue with the single file format for now to keep things simple.

The **Global** portion of this app is super simple, we are just loading the ‘shiny’ package into our environment.

The **UI** portion here actually has a little ore going on. As you can see, this portion is just one big function called fluidPage() with a few other things happening inside. The visual portion of shiny is all based off HTML, the coding language used to build websites. If you are curious, right click on any webpage and hit the ‘inspect’ button in the menu and it will show you all the underlying HTML. It’s really gross!! Shiny is cool because it allows you to wrangle the flexibility and power of HTML, without needing to know how to code in HTML. Shiny provides you with a surprisingly comprehensive set of functions that build HTML layouts as a result so you don’t have to. Phew!!

So, lets break down whats happening in the fluidPage() function. There are many nested function within it, but the next immediate level down is sidebarlayout(). This function is a pre-packaged layout that has a side bar and main body areas you can put stuff into. The two functions at the next level down are sidebarPanel() and mainPanel(). These tell R what you want to put into each of these areas. In sidebarPanel() we have a sliderInput(), which creates a slider bar the user can interact move back and forth that we internally are calling ‘bins’ (I’ll explain what that means in a bit), we gave it a title to display above the slider bar ‘Number of bins:’, and restricted the values between 1 and 50, and set the initial value to 30. Inside the mainPanel() function we have plotOutput(), which, unlike an input function, just creates a spot something can be put later. In this case, this spot will only accept a plot. We named this currently unoccupied spot ‘distPlot’.

If we run this code, becuse it is a single file type app and it is already structured properly, things will run from top to bottom. Starting with loading the package.