Shiny Apps in R

Matt Hewitt

2024-08-21

Welcome to a help/info guide for R shiny!  
  
R shiny is a package for program R that allows you to build applications with an HTML front end, and an R backend. That means that the portion of the app that a user sees is built using bits of HTML code pre-packaged into R functions that you can use to build user interfaces (UI). The back end of a shiny app, or server, operates using R code. It is a super useful tool that allows you to bring complex data processing or other complex operations to folks that use the app.

I started learning about shiny with the intention of having a front end interface that allows technicians to enter data to Borderlands Research Institute’s (BRI) data warehouse, an Amazon Web Service (AWS) web hosted PostgreSQL relational database, without learning how to code in R or the need to interact with the BRI warehouse.

These ambitions led me to build two applications. A bear project management app where members of the bear team, students and technicians, can view and upload data, manage some basic inventory (collars, inReach devices, cameras), and even view properties and collar data on an interactive map.  
The other is an application intended to ease the workload for camera trap projects. It integrates Microsoft’s open source megadetector image recognition model to identify whether or not there is an animal in a photo, and a user interface for technicians to identify detections to species. After processing, the data is submitted to the BRI data warehouse with photos being stored in the BRI AWS S3 bucket. Another web hosted data storage system that works perfect for large data types.

# Shiny Structure

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

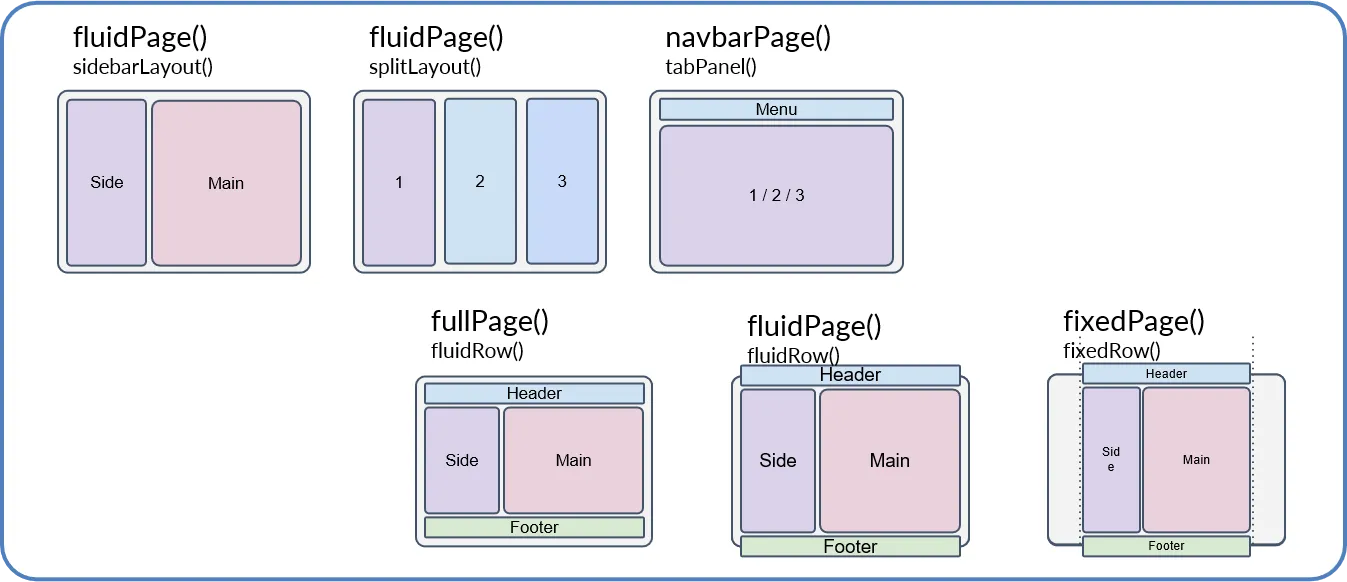
### Global

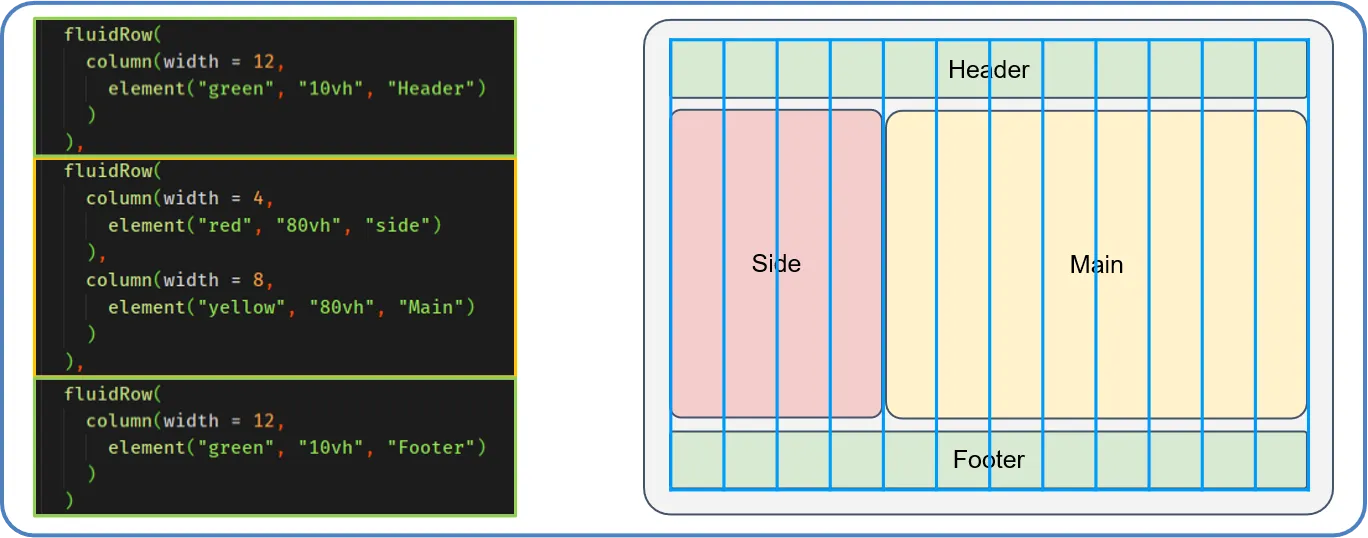
This part of the app is intended for you to ‘get things set up’. This is where you would load all the necessary packages, load data needed for app operation, connect to databases, source other files/scripts, set up reactive objects, and any other pieces of information your app needs to operate. This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

### UI (User Interface)

This is the portion of the app the user actually sees and interacts with. It can be as specific or as general as needed. For apps with little function or simple layouts, all layouts can be specified in the UI. But if you are planning on having a very dynamic layout, only the bare bones of the layout would be specified in the UI, and all other dynamic parts will be built in the server file.

There are many pre designed layouts available to start from. There are a few that come with the shiny package (see below), but there are some other packages that provide more 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 building 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 for an example 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 elements on the page. Elements can be anything, user inputs, outputs, tables, figures, boxes, titles, anything you want to place on the page. The method for ‘mapping’ where items are placed follow this structure:  


This graphic is a custom made layout, not one of the preset layouts through shiny. The three sections in the R code on the left side (the three fluidRow()’s) correspond to the three rows on the example layout on the right (the Header, Side/Main, and Footer). The first fluidRow() that creates the Header section contains a column() function. Just as the names would suggest, fluidRow() creates rows, and column() creates columns. Because column() is nested within fluidRow() in this instance, it will create a single column of width 12 within the Header section. It is within column() that you would place elements to have them be displayed within the Header section.

You can see the center portion of the code has two column()’s within the fluidRow(). These correspond to the Side and Main portions of the layout on the right. The first column() is of width 4, and the second column() is of width 8, which add up to the total width of 12 to fill the width of the page. It isn’t shown explicitly in this figure, but the width of each section is 12. So even though the Side section is only of with 4 when looking at the entire page, when we start placing elements within the section, we have a width of 12 to work with again.

The height of the different sections is dictated by what elements you place within each column(). If you ran this example code here, all three fluidRow()’s would be the same height.

### Server

This is where the magic happens! The **Server** file 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.

There are two parts, essentially list objects, that live underneath the operations of the server; the input and output. The input object is where all element input values live. Examples of input elements are textInput(), selectInput(), actionButton(), numericInput(), and dateInput(). These are very handy input functions that create elements users can interact with to input data.

When you create an element in the **UI** (or wherever you create the element) you assign it an inputid. This id, or tag, will be how you access the value of, or make functions observe, different input elements.

A major part of how shiny operates is by *observing* different portions of the **Server** file. There are many functions that dictate how this works, but some of the most common ones are observe() and observeEvent(). The observe() function tells R to continuously run the code within it. I don’t entirely understand how R knows when or how often to run all the observe() functions within the **Server** file, but somehow it gets it right. The code within observeEvent() functions will only be run when a specific event happens.

# Shiny Function

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 is one of the pre-packaged base shiny layouts that has a side bar and main body areas we saw above. The two functions at the next level down are sidebarPanel() and mainPanel(). This is where you 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 move back and forth that we internally labeled as ‘bins’ (I’ll explain what that means in a bit), gave it a title to display above the slider bar ‘Number of bins:’, 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 internally labeled this currently unoccupied spot ‘distPlot’.

If we run this code, because 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.