R Shiny

Daniel J. Eck

We're going to build Shiny apps

Open the Rmd file for the notes on your local computer to interact with the Shiny app.

What are shiny apps? Why should we use them?

These are free, easy-to-use, almost-easy-to-create, interactive data products. Shiny apps represent digestible content for anyone to learn about statistics and data through direct manipulation of the data values. The nice thing is that the app is **reactive** and changes with every confirmed change the user (of the app) makes.

How to build a Shiny app?

There are some basic functions and syntax that is common to all shiny apps, while other functions and tasks will be specific to the type of data product you want to show.

Every Shiny app must have 3 components:

1. **ui** or user interface - this component is where the appearance, user inputs, and resulting outputs are organized and displayed.
2. **server** function (followed by grouped expression{}) - this component is where reactive calculations from user inputs are conducted
3. **shinyApp** function - this component knits the ui and server functions together.

Let's begin by looking at the coding for the the Faithful app (the template is in RStudio already). This app depicts the waiting times to the next eruption of the Old Faithful geyser.

First load in the shiny library.

```
library(shiny)
```

We see that the **ui** function for the Old Faithful app has a slider input feature and a plot.

```
# Define UI for application that draws a histogram
ui <- fluidPage(

    # Sidebar with a slider input for number of bins
      sliderInput(inputId = "bins",
                          label = "Number of bins:",
                          min = 1,
                          max = 50,
                          value = 30), #the comma is important

    # Show a plot of the generated distribution
      plotOutput("distPlot")
)
```

We see that the **server** function for the Old Faithful app first takes the bins input, constructs equally spaced bins to be used as an input as a histogram, and then builds the histogram.

```
# Define server logic required to draw a histogram
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')
    })
}
```

The **shinyApp** function puts it all together so that user input is then displayed as output.

```
# Run the application
shinyApp(ui = ui, server = server)
```

We can put it all together in a code chunk which allows one to run this shiny app.

```
library(shiny)

# Define UI for application that draws a histogram
ui <- fluidPage(

    # Sidebar with a slider input for number of bins
      sliderInput(inputId = "bins",
                        label = "Number of bins:",
                        min = 1,
                        max = 50,
                        value = 30), #the comma is important

    # Show a plot of the generated distribution
      plotOutput("distPlot")
)

# Define server logic required to draw a histogram
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')
    })
}

# Run the application
shinyApp(ui = ui, server = server)
```

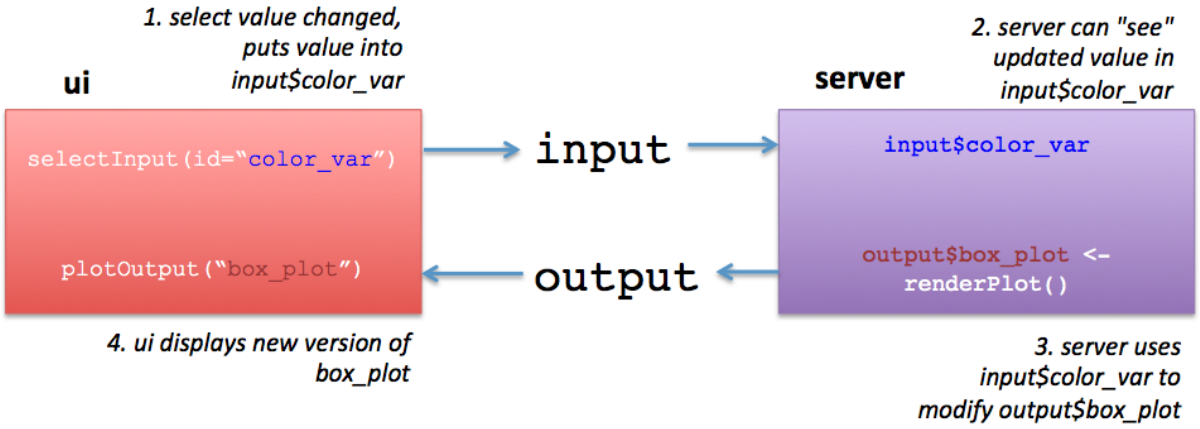

Figure 1: Image source

Further details on shiny components

ui

The user interface contains code that builds the web document by creating html for the app. This ui object contains the `fluidPage()` function which contains all the "layers" of the design of the app. The app will need **input** and **output** designs.

There are several **input** designs:

- buttons `actionButton()` or `submitButton()`
- single or group checkboxes `checkboxInput()` or `checkboxGroupInput()`
- date input `dateInput()` or range `dateRangeInput()`
- file input `fileInput()`
- numeric input `numericInput()`
- text input `textInput()`
- radio buttons `radioButtons()`
- select box (i.e. dropdown menu) `selectInput()`
- sliders `sliderInput()`

For these input designs we need an `inputId` for identifying the input (and for use with the server function) and a `label` or explanation of the input type. Users will read this label. The graphic below displays the appearance of several of these input designs.

There are several **output** designs:

- interactive table `dataTableOutput()`
- raw html `htmlOutput()`
- image `imageOutput()`
- plot `plotOutput()`
- table `tableOutput()`
- text `textOutput()` or `verbatimTextOutput()`

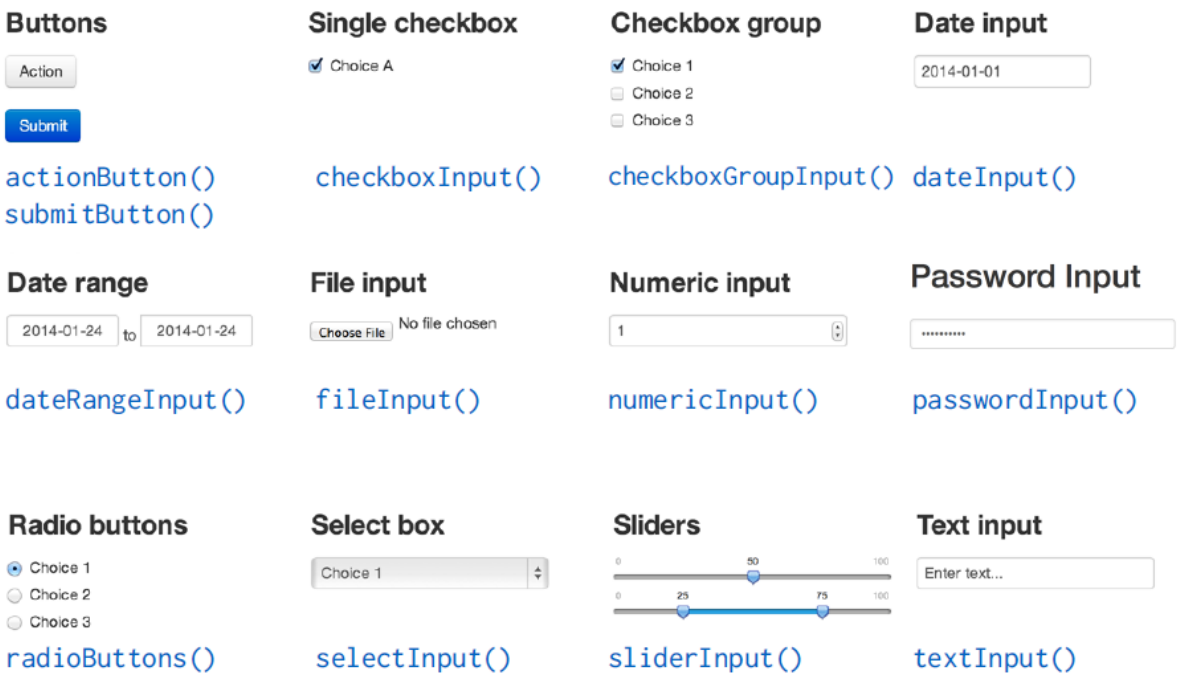

Figure 2: Image source

- a Shiny UI element `uiOutput()`

For these output designs, we need an `outputId`. None of the fluidPage arguments will create any meaningful output for us to see. It does create html for the page.

server()

The `server()` function is the R code (that we're used to) giving instructions for the inputs and outputs. The `server()` function creates/builds/re-builds the **output** of the app such as the plot, table, and text by first recognizing the **input** (the data values). The `server()` function must have both **input** and **output**.

The **input** in the `server()` function must be named as `input$*` where the asterisk represents the naming of the object which should match the `inputId` (created in the ui portion of the code). the input is a list.

If building an **output** object, it needs to be saved as `output$*` where the asterisk represents the naming of the object which should match the `outputId` (created in the ui portion of the code). The output is a list.

To display the output object, we use the `render*()` function, where the asterisk is a particular type of render function. Usually the `render*()` function has a corresponding output type (see ui above).

- interactive table `renderDataTable()`
- image `renderImage()`
- plot `renderPlot()`
- a code block `renderPrint()`
- table `renderTable()`
- character string `renderText()`
- a shiny UI element `renderUI()`

Reactivity When a user changes an input value in the app and the app output changes as a result, that is called **reactivity**.

If we want the **output** (histogram in the above app) to change after the user changes the **input** (slider in the above app), then we need to use the `input$*` matching the Id of the input in the ui and that input must go insde the `render*()` function. *If we do this correctly, then the reactivity occurs automatically!*

Reactive values (i.e. the `input$`) work together with **reactive functions.**

shinyApp()

The `shinyApp()` function serves as the "knitting" function to weave together the ui and `server()` function. This function creates the app which is running locally on your computer.

Further Resources

I have included my "The Challenging Nostalgia and Performance Metrics in Baseball Project" Shiny app contents in the same directory as these notes.

You can watch Parts 1-3 of the Shiny app tutorial by RStudio to learn more on your own. The following notes are from Grolemund's video and slides https://github.com/rstudio-education/shiny.rstudio.com-tutorial.

You can also view David Dalpiaz's R Shiny resources for an app that he made for STAT 385, and R shiny resources from when I taught STAT 385.

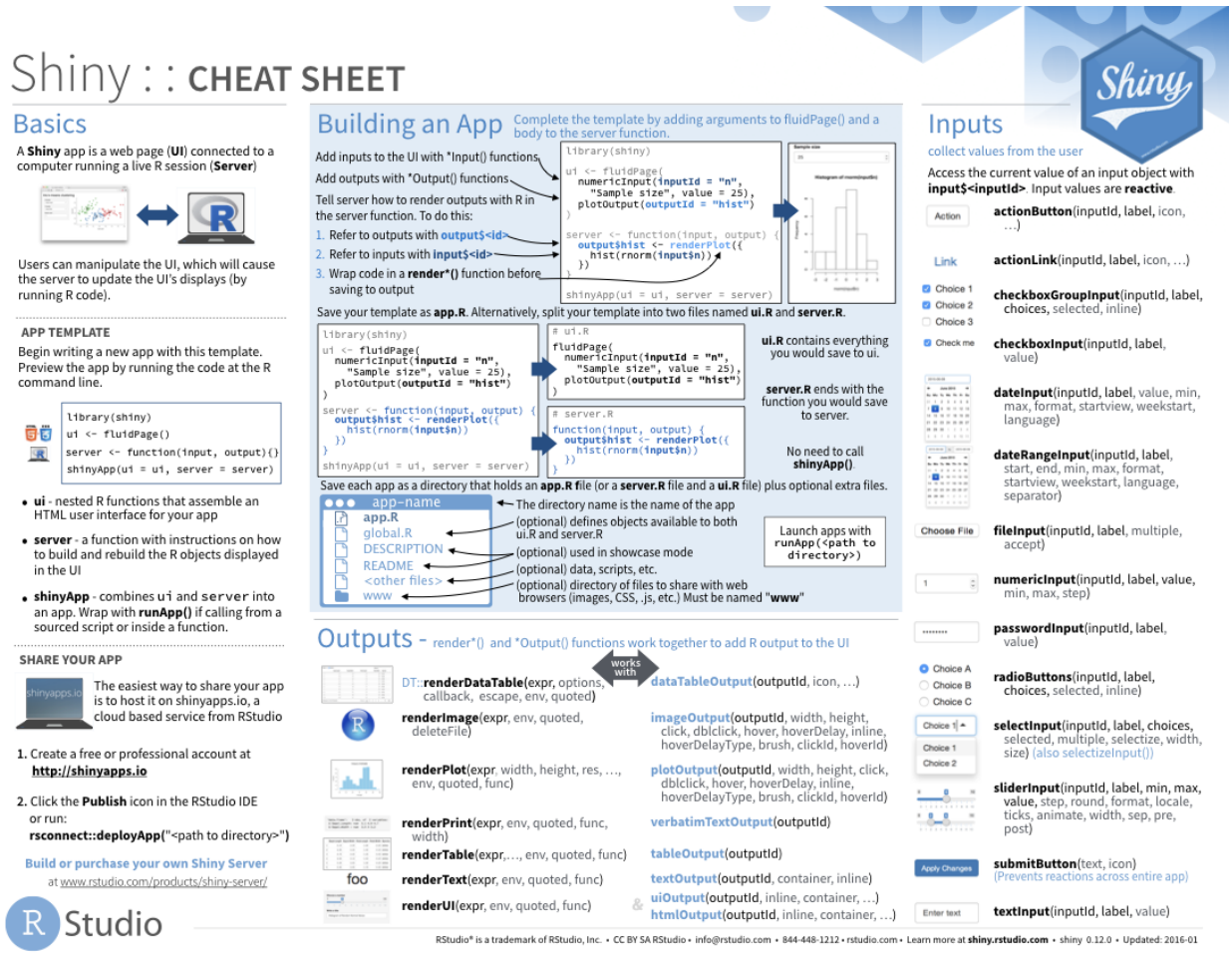

Figure 3: Image source