Paper DV06

Building interactive web apps – Experiences with R Shiny

Stephanie Fechtner, Bayer AG, Wuppertal, Germany

ABSTRACT

The open source R package Shiny provides an opportunity to build interactive web applications in an efficient way by using the statistical software R and the user interface RStudio; no web development skills are required. The created web application can be accessed via the internet; the user of the application himself does not have to use or to know R. However, the more interactive the tool should be in the end, the more hurdles need to be taken by the R programmer. The programmer needs to understand the purposes of the two components of a Shiny app - the user interface object and the server function - and how they can affect each other and the appearance of the application. With some experience it is possible, to create smoothly working Shiny apps which can make the exploration of clinical data easier and the presentation of statistical analyses more attractive.

INTRODUCTION

A picture says more than thousand words. But how can a good picture be created, if the data situation is not clear or if it changes quite often? Different people could be interested in various figures which might be similar; but nevertheless, a programmer would need to create a new figure every time, if requested. If the data situation is not clear, a lot of uninteresting figures would need to be produced until it is clear which figures are really needed; these unneeded figures require capacity to be stored somewhere. Or the other way around: if a study or project team sticks to producing standard tables and figures, possible relationships between variables could be overseen.

Medical experts, physicians and investigators have the expertise to know which correlations might be of interest in a specific study or project but they do not know in advance, what the data is going to tell; and usually they do not feel comfortable to program their own figure, especially not in the statistical programming languages R or SAS.

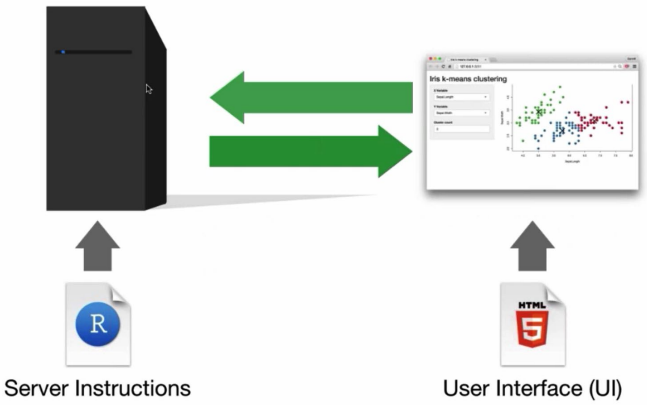
The application R Shiny could be one option for a programmer or statistician to give the investigators the opportunity to explore the data by themselves without knowing the program R. Shiny is an R package that creates a web application which can be accessed via the internet. These apps can reduce the output of unused figures and tables and new data aspects can be easily investigated by medical experts or physicians.

In this paper, it will be explained, how R Shiny works, which requirements need to be fulfilled before the programmer can start to create an app and how a Shiny app can be built. In this context, an own created app will be shown, to explain, how the R program affects the appearance and the output of the app.

R SHINY

To build a Shiny app, the programmer needs to have installed the statistical programming language R version 3.0.2 or higher as well as the integrated development environment RStudio. The Shiny programmer needs to know at least how to program in R and how to use RStudio. Furthermore, it is necessary to install the R package `shiny` before working on the Shiny app.

Every Shiny app is maintained by a computer running R and consists of two components: the user interface and the server instructions. The user interface is a web page that the app user will see; it is a web document written HTML although the programmer of the app do not need to know any HTML because the document will be generated from R. The server instructions is a set of instructions which will tell the server what to do if the user of the app changes the different input objects within the app.



In the following section, the creation of the user interface and the server function will be explained, how they affect each other and how the thoughtful creation of these two components can result in a smoothly working web app.

HOW TO BUILD A SHINY APP – STEP BY STEP

If the programmer wants to start to build his Shiny app, it is recommended to always start with the same template:

```
library(shiny)
ui <- fluidPage()
server <- function(input, output) {}
shinyApp(ui = ui, server = server)
```

The first line of code sets up the user interface object (the ui-object), the second line of code defines the server object and the last one adds the two components together into a Shiny app.

THE USER INTERFACE

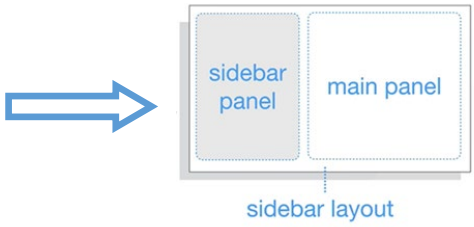
The user interface is the web page that the user can see when he uses the app. It consists of input and output objects which should be in mind of the programmer who builds the Shiny app. Input objects can be modified by the user to provide any kind of values to the app; outputs are R objects that the user can see (plots, tables, text,…) and which will respond if the user changes the inputs.

The programmer can add inputs and outputs to the `fluidPage()` of the template:

```
ui <- fluidPage(
       # *Input() functions,
       # *Output() functions
)
```

When the programmer starts to build the Shiny app, it is worth to think about where to display these input and output objects. The `sidebarLayout()` can be used to quickly create a sidebar design with `sidebarPanel()` and `mainPanel()` which divide the app into two sections:

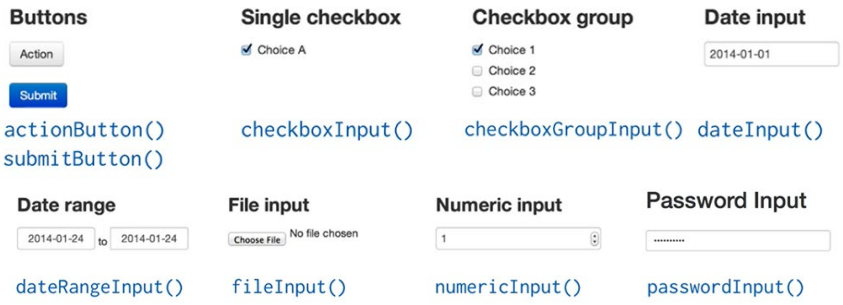
```
ui <- fluidPage(
  sidebarLayout(
   sidebarPanel(),
   mainPanel()
  )
)
```

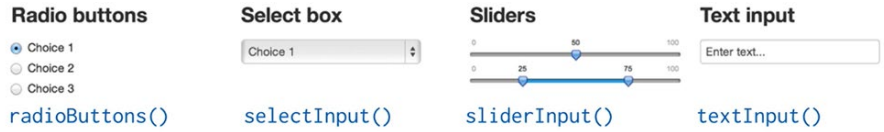


The widths of these panels can be chosen by the programmer. The sidebarPanel can be used to place the input objects and the main panel to place the output objects.

Input functions

Shiny provides several input functions which can be used inside the app:





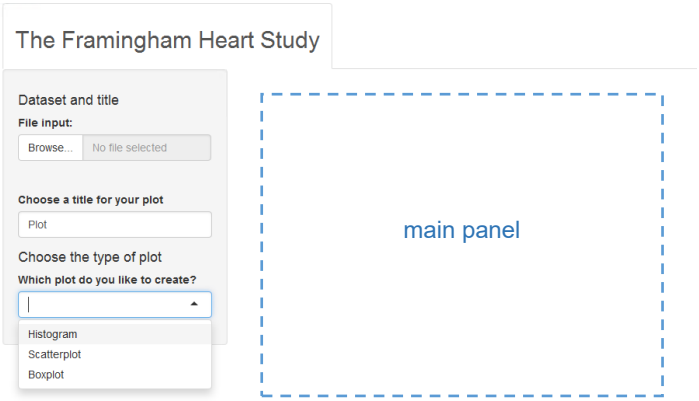
All of these input functions need specific arguments. The `inputId` and `label` argument are required for all objects to give the input function a unique name and to display a label which the user can see. Furthermore each of these input objects needs specific arguments which are explained on the R help page for each input object.

Example – The Framingham Heart Study

To allow the user to investigate the "Heart"-dataset of the Sashelp-library, an R Shiny app can be created. This dataset contains the data of the first 5209 adult patients that were included in the Framingham Heart Study and 17 variables (Status, Cause of Death, Age CHD Diagnosed, Sex, Age at Start, Height, Weight, Diastolic, Systolic, Metropolitan Relative Weight, Smoking, Age at Death, Cholesterol, Cholesterol Status, Blood Pressure Status, Weight Status, Smoking Status). The aim of this study was to detect causes and risks of the Coronary heart disease (CHD) (SAS, 2017).

To upload the heart-dataset within the app, a `fileInput()`-object should be provided in the beginning of the `sidebarPanel`. To give the user the opportunity to decide, which kind of plot, he would like to see, an `selectInput()`-object is provided, where the user is able to choose between a histogram, scatterplot or boxplot. Furthermore, an additional `textInput()`-object was created to allow the user to name his plot the way he wants to.

```
ui <- fluidPage(
  tabsetPanel(
   tabPanel(titlePanel("The Framingham Heart
                        Study"),
    sidebarLayout(
     sidebarPanel(width=2,
      h4("Dataset and title"),
      fileInput("file", "File input:", accept =
                c("text/csv", ".csv")),
      textInput("title", label="Choose a title
                for your plot", value="Plot"),
      h4("Choose the type of plot"),
      selectInput(inputId="Plot_type",
        label="Which plot do you like to
        create?", list('','Histogram',
        'Scatterplot', 'Boxplot'),
        multiple=FALSE)
     ),
     mainPanel()
    )
   )
  )
)
```



With this "heart-app" it is now possible for the user to define the first inputs, but no output will be created at the moment. First, the app needs to know, which kind of output objects should be created and where they should be placed.

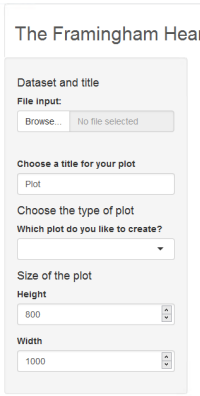
Output functions

Within the mainPanel the chosen output can be created. To this end, several output functions are available in Shiny:

Function	Inserts
dataTableOutput()	an interactive table
htmlOutput()	raw HTML
imageOutput()	Image
plotOutput()	Plot
tableOutput()	Table

textOutput()	Text
uiOutput()	A Shiny UI element
verbatimTextOutput()	text

Like the input functions, the output functions also need different arguments. A required argument is called `outputId` which is necessary to give the output object a unique name.
For the heart-app, a plot should be created. Therefore a `plotOutput()`-statement is added within the main panel. To give the user the control about the height and the width of the plot, a `height` and a `width` argument were also added within `plotOutput()`. In this context, `numericInput()`-functions were added to the sidebarPanel to provide the corresponding input objects:




```
ui <- fluidPage(
  tabsetPanel(
   tabPanel(titlePanel("The Framingham Heart Study"),
    sidebarLayout(
     sidebarPanel(width=2,
      h4("Dataset and title"),
      fileInput("file", "File input:", accept =
                c("text/csv", ".csv")),
      textInput("title", label="Choose a title for
                your plot", value="Plot"),
      h4("Choose the type of plot"),
      selectInput(inputId="Plot_type", label="Which
       plot do you like to create?", list('',
       'Histogram', 'Scatterplot', 'Boxplot'),
       multiple=FALSE),
      h4("Size of the plot"),
      numericInput(inputId="hei", label="Height", value=800, min=500, max=2000),
      numericInput(inputId="wid", label="Width", value=1000, min=500, max=2000)
     ),
     mainPanel(width=10,
      plotOutput(outputId = "plot", height="auto", width="auto")
     )
    )
   )
  )
)
```

The app provides now the defined input and output objects, but is not yet able to create the desired output object. The problem is, that the app just knows, that a plot with the outputId `plot` should be created within the main panel, but it does not know how to create this object. This will be done within the server function which will be explained in the next section.

THE SERVER FUNCTION

The server function consists of a set of instructions which will tell the server how to assemble inputs into outputs.
To use the server function the programmer needs to follow 3 rules:

1. Save objects to display to `output$`: `output$object` → `plotOutput("object")` (use the same name in both components)
2. Build objects to display with `render*()`: creation of the desired type of output. Several render-functions are available in Shiny:

Function	creates
renderDataTable()	An interactive table (from a data frame, matrix or other table-like structure)
renderImage()	An image (saved as a link to a source file)
renderPlot()	A plot
renderPrint()	A code block of printed output
renderTable()	A table (from a data frame, matrix or other table-like structure)
renderText()	A character string
renderUI()	A Shiny UI element

Within the brackets of each function the programmer defines what object should be built by writing an R code block that builds the object; the programmer can write as many lines of R code between the brackets as he wishes or needs.

3. Use input values with `input$` when the outputs are created: `inputId` = "`object`" → `input$object`

For the heart-app a plot should be created dependent on the choice, the user has made. However, some steps need to be programmed before. First of all, the uploaded heart-dataset should be saved in an R object within the server function, so that this data can be used for the creation of the desired plot:

```
server <- function(input, output) {
  data1 <- reactive({
    if (is.null(input$file)){
      return(NULL)
    }
    read.csv(input$file$datapath, header=T, sep=",")
  })
}
```

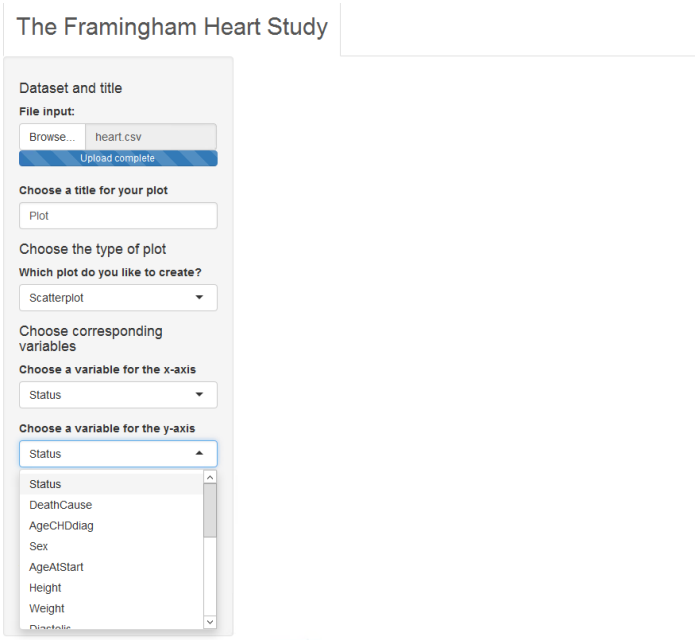
The `reactive()`-statement is used to ensure, that the same dataset, once it has been uploaded, can be used the whole time. For further insights on reactivity, refer to the R Shiny Tutorial.
After the upload of the dataset, the user can decide which plot he would like to create. But for the creation of the plot he also needs to decide which of the 17 different variables of the Heart-dataset should be displayed within the graph, and the corresponding columns within the dataset should be used for the plot creation. This choice can be made after the dataset was uploaded. The column names of the dataset should be then available as the options for the definition of the x- and y-axis. This can be done with the aid of the `renderUI()`-statement within the server function as follows:

```
server <- function(input, output) {
  data1 <- reactive({
    if (is.null(input$file)){
      return(NULL)
    }
    read.csv(input$file$datapath, header=T, sep=",")
  })
  output$x <- renderUI({
    if (is.null(data1())) return()
    tagList(
      selectInput(inputId="x", label="Choose a variable for the x-axis", names(data1()),
        multiple=FALSE)
    )
  })
  output$y <- renderUI({
    if (is.null(data1()) || input$Plot_type == "Histogram") return()
    tagList(
      selectInput(inputId="y", label="Choose a variable for the y-axis", names(data1()),
        multiple=FALSE)
    )
  })
}
```

In this context, the user interface also needs to be changed to display the input functions x and y after the dataset is uploaded:

```
ui <- fluidPage(
  tabsetPanel(
   tabPanel(titlePanel("The Framingham Heart Study"),
    sidebarLayout(
     sidebarPanel(width=2,
      h4("Dataset and title"),
...
      selectInput(inputId="Plot_type",
       label="Which plot do you like to
       create?", list('', 'Histogram',
```

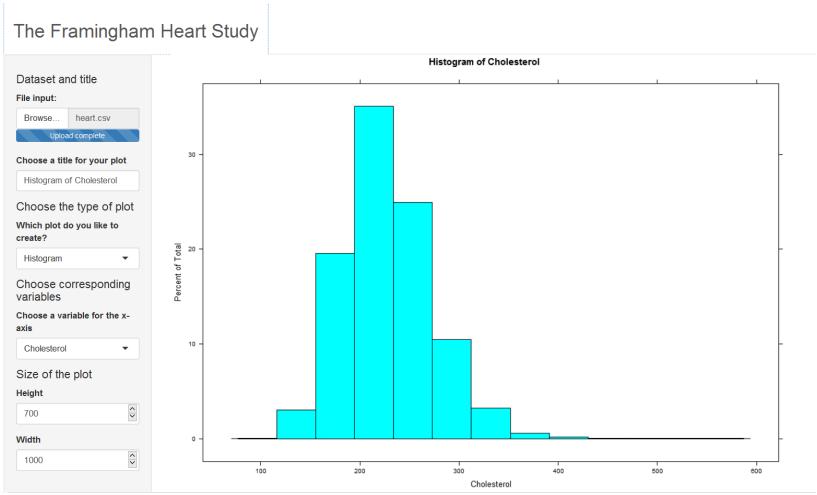
```
       'Scatterplot', 'Boxplot'),
       multiple=FALSE),
      h4("Choose corresponding variables"),
      uiOutput("x"),
      uiOutput("y"),
...
)
```



The input functions for x and y will appear as soon as the user uploads a dataset with the correct format. The input function for y disappears again, if the user chooses to create a histogram, because it is not needed in this case.

Now, the plot output can be created within the server function by using `renderPlot({})`. Between the curly brackets, the programmer can add as much code as he wants. To create the plot dependent on the choice of the user, the programmer could use the following code:

```
server <- function(input, output) {
...
  output$plot <- renderPlot({

    if (is.null(data1())) return()

    # Histogram #
     if (input$Plot_type == "Histogram") {
      histogram( ~ data1()[,which(names(data1())==input$x)], data=data1(), xlab=input$x,
                main=input$title)
    }


    # Scatterplot #
    else if (input$Plot_type ==
        "Scatterplot") {
        xyplot(data1()[,which(names(data1())=
        =input$y)] ~
        data1()[,which(names(data1())==input$
        x)], data=data1(), xlab=input$x,
        ylab=input$y, main=input$title)
    }

    # Boxplot #
    else if (input$Plot_type == "Boxplot") {
        bwplot(data1()[,which(names(data1())=
        =input$y)] ~
        data1()[,which(names(data1())==input$
        x)], data=data1(), xlab=input$x,
        ylab=input$y, main=input$title)
    }

  }, height=function(x) input$hei, width=function(x) input$wid) # Height and width of the plot

}
```



Now, dependent on the choice of the user, a histogram, scatterplot or boxplot of the chosen variable(s) appears in the main panel together with the title entered in the text field. The inputs change the appearance of the outputs automatically after the changes are made. To give the user the choice to update the plot at the time he wants to, it would also be possible to add an `Update`-button within the sidebarPanel; then the plot only changes when the user clicks on this button.

FURTHER FUNCTIONS/ APPEARANCE OF THE APP

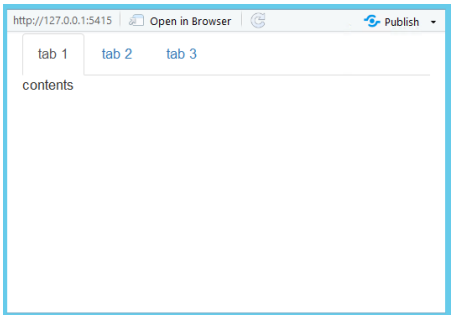
There are a lot of possible ways to change the appearance of the app. Pictures or links to web pages can be incorporated and text in different styles as well as space between rows and/or columns can be added into the app. Elements can be positioned in a grid or stacked in layers.

The use of layers of panels could be quite helpful, if the programmer wants to add more tables, data or plots to the app. In this way, it would be e.g. possible, to add more plots or additional information to the created graph in different panels. The different kind of panels that Shiny provides are:

Panel	Description
absolutePanel()	Panel position set rigidly (absolutely), not fluidly
conditionalPanel()	A JavaScript expression determines whether panel is visible or not
fixedPanel()	Panel ist fixed to browser window and does not scroll with the page
headerPanel()	Panel for the app's title, used with pageWithSidebar()
inputPanel()	Panel with grey background, suitable for grouping inputs
mainPanel()	Panel for displaying output, used with pageWithSidebar()
navlistPanel()	Panel for displaying multiple stacked tabPanel(). Uses sidebar navigation
sidebarPanel()	Panel for displaying a sidebar of inputs, used with pageWithSidebar()
tabPanel()	Stackable panel. Used with navlistPanel() and tabsetPanel()
tabsetPanel()	Panel for displaying multiple stacked tabPanels(). Uses tab navigation
titlePanel()	Panel for the app's title, used with pageWithSidebar()
wellPanel()	Panel with grey background

Panels group elements into a single unit for aesthetic or functional reasons. `tabPanel()` can be used to create a stackable panel and `tabsetPanel()` to arrange tab panels into a stack with tab navigation:

```
fluidPage(
  tabsetPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")
  )
)
```

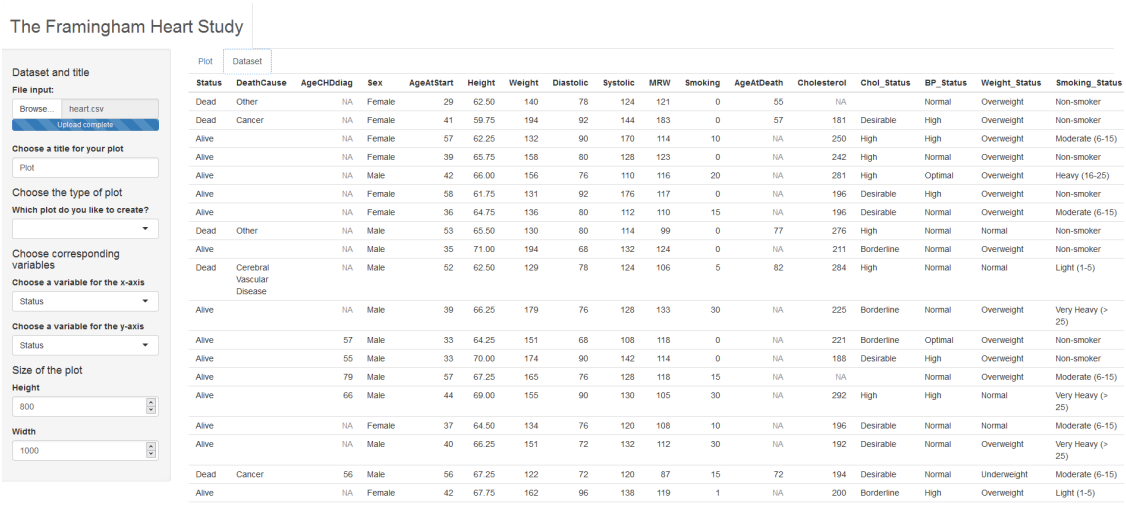


For the heart-app an additional tab was created, so that the user also has the possibility to see the heart-dataset after the upload. Within the user interface, the `mainPanel()` was updated as:

```
mainPanel(width=10,
  tabsetPanel(type = "tabs",
    tabPanel("Plot", plotOutput(outputId = "plot", height="auto", width="auto")),
    tabPanel("Dataset", tableOutput("dataset"))
  )
)
```

The corresponding dataset was created within the server function by using `renderTable()`:

```
output$dataset <- renderTable({
  data1()
})
```

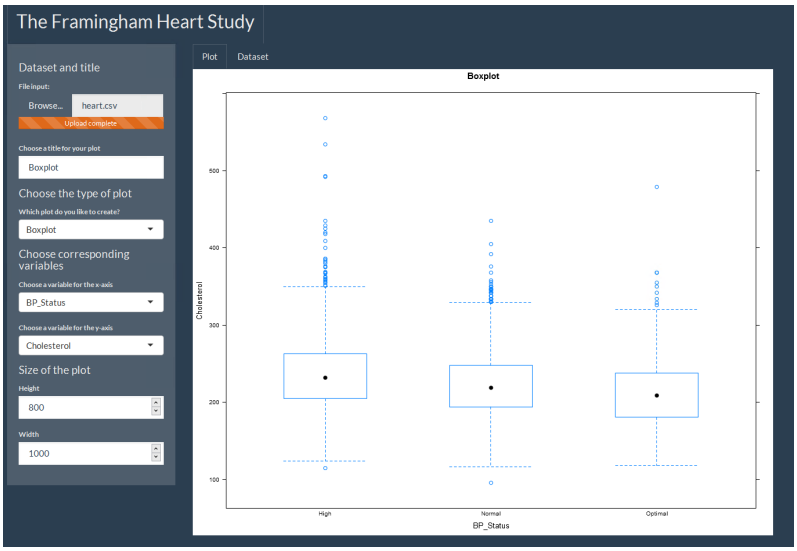


Status	DeathCause	AgeCHDdiag	Sex	AgeAtStart	Height	Weight	Diastolic	Systolic	MRW	Smoking	AgeAtDeath	Cholesterol	Chol_Status	BP_Status	Weight_Status	Smoking_Status
Dead	Other	NA	Female	29	62.50	140	78	124	121	0	55	NA		Normal	Overweight	Non-smoker
Dead	Cancer	NA	Female	41	59.75	194	92	144	183	0	57	181	Desirable	High	Overweight	Non-smoker
Alive		NA	Female	57	62.25	132	90	170	114	10	NA	250	High	High	Overweight	Moderate (6-15)
Alive		NA	Female	39	65.75	158	80	128	123	0	NA	242	High	Normal	Overweight	Non-smoker
Alive		NA	Male	42	66.00	156	76	110	116	20	NA	281	High	Optimal	Overweight	Heavy (16-25)
Alive		NA	Female	58	61.75	131	92	176	117	0	NA	196	Desirable	High	Overweight	Non-smoker
Alive		NA	Female	36	64.75	136	80	112	110	15	NA	196	Desirable	Normal	Overweight	Moderate (6-15)
Dead	Other	NA	Male	53	65.50	130	80	114	99	0	77	276	High	Normal	Normal	Non-smoker
Alive		NA	Male	35	71.00	194	68	132	124	0	NA	211	Borderline	Normal	Overweight	Non-smoker
Dead	Cerebral Vascular Disease	NA	Male	52	62.50	129	78	124	106	5	82	284	High	Normal	Normal	Light (1-5)
Alive		NA	Male	39	66.25	179	76	128	133	30	NA	225	Borderline	Normal	Overweight	Very Heavy (> 25)
Alive		57	Male	33	64.25	151	68	108	118	0	NA	221	Borderline	Optimal	Overweight	Non-smoker
Alive		55	Male	33	70.00	174	90	142	114	0	NA	188	Desirable	High	Overweight	Non-smoker
Alive		79	Male	57	67.25	165	76	128	118	15	NA	NA		Normal	Overweight	Moderate (6-15)
Alive		66	Male	44	69.00	155	90	130	105	30	NA	292	High	High	Normal	Very Heavy (> 25)
Alive		NA	Female	37	64.50	134	76	120	108	10	NA	196	Desirable	Normal	Normal	Moderate (6-15)
Alive		NA	Male	40	66.25	151	72	132	112	30	NA	192	Desirable	Normal	Overweight	Very Heavy (> 25)
Dead	Cancer	56	Male	56	67.25	122	72	120	87	15	72	194	Desirable	Normal	Underweight	Moderate (6-15)
Alive		NA	Female	42	67.75	162	96	138	119	1	NA	200	Borderline	High	Overweight	Light (1-5)

In addition to this layout-function, it is also possible to change the overall appearance of the app by using the R-package `shinythemes` and to add the selected theme at the beginning of the program for the user interface:

```
install.packages("shinythemes")
library(shinythemes)

fluidPage(theme =
          shinytheme("superhero"),…)
```

A list of all available themes can be found here.
Alternatively, the programmer can add a theme-selector in the user interface to quickly change the style of the app by using `Shinythemes::themeSelector()` instead of `theme = shinytheme(…)`.



THE HEART-APP – COMPLETE CODE

This section provides the whole R code which was programmed to create the heart-app. In addition to the features explained above, the user has furthermore the option to display different aspects of each plot:

- If "Histogram" is chosen, the user has the chance to decide, if he wants to see the percentage, the counts or the distribution of the data.
- If "Scatterplot" is chosen, the user should decide, if he wants to display the plot with further consideration of a group variable or not.
- If "Boxplot" is chosen, the user can decide about the ratio of the boxes (between 0 and 1).

The user is just able to see these inputs which are necessary for the chosen plot type; e.g. no selection about the grouping variable should be made, if "Histogram" or "Boxplot" is selected. This problem was solved by using further `uiOutput`-statements within the user interface and `renderUI({})` within the server function.
A `submitButton` was also added to the user interface to give the user the control on when the output should be updated according to the chosen inputs. In addition, further variable(s) can be chosen as additional splitting variable(s) for each of the plots:

```
library(lattice)
library(shiny)
library(shinythemes)

ui <- fluidPage(theme = shinytheme("flatly"),
  tabsetPanel(
   tabPanel(titlePanel("The Framingham Heart Study"),
    sidebarLayout(
     sidebarPanel(width=2,
      h4("Dataset and title"),
      fileInput("file", "File input:", accept = c("text/csv", ".csv")),
      textInput("title", label="Choose a title for your plot", value="Plot"),  ## Plot title ##
      h4("Choose the type of plot"),                                    ## Text in sidebarPanel ##
      selectInput(inputId="Plot_type", label="Which plot do you like to create?", list('',
      'Histogram', 'Scatterplot', 'Boxplot'), multiple=FALSE), ## Which plot type to create? ##
      h4("Choose corresponding variables"),                             ## Text in sidebarPanel ##
      uiOutput("x"),                                          ## Choose variable for the x-axis ##
      uiOutput("y"),                                          ## Choose variable for the y-axis ##
      uiOutput("hist_type2"),                      ## Percent, count or density for histogram? ##
      uiOutput("group2"),                                        ## Choose grouping variable ##
      uiOutput("box_ratio"),                                    ## Choose ratio of the boxes ##
      uiOutput("split"),                           ## Choose (optional) splitting variable(s) ##
      h4("Size of the plot"),                                   ## Text in sidebarPanel ##
      numericInput(inputId="hei", label="Height", value=800, min=500, max=2000),   ## Height ##
      numericInput(inputId="wid", label="Width", value=1000, min=500, max=2000),   ## Weight ##
      submitButton("Update!")
     ),
     mainPanel(width=10,
      tabsetPanel(type = "tabs",
       tabPanel("Plot", plotOutput(outputId = "plot", height="auto", width="auto")),
       tabPanel("Dataset", tableOutput("dataset"))
     )
    )
   )
  )
 )
)


server <- function(input, output) {

  ####### Read the dataset -> csv-format required #######
  data1 <- reactive({
    if (is.null(input$file)){
      return(NULL)
    }
    read.csv(input$file$datapath, header=T, sep=",")
  })


  ######### Choose variables which should be displayed within the plot ##################
  ## Which variable should be displayed on the x-axis? --> necessary for each plot ##
  output$x <- renderUI({
    if (is.null(data1())) return()
    tagList(
      selectInput(inputId="x", label="Choose a variable for the x-axis", names(data1()),
                  multiple=FALSE)
    )
  })
  ## Which variable should be displayed on the y-axis? --> not necessary for Histogram ##
  output$y <- renderUI({
    if (is.null(data1()) || input$Plot_type == "Histogram") return()
    tagList(
      selectInput(inputId="y", label="Choose a variable for the y-axis", names(data1()),
                  multiple=FALSE)
    )
  })
  ## Percent, count or density for histogram? --> Just necessary if Plot_type=Histogram ##
```

```
output$hist_type2 <- renderUI({
  if (is.null(data1()) || input$Plot_type != "Histogram") return()
  tagList(
    radioButtons(inputId="hist_type", label="Display of percent, count or density?",
                 choices=list('Percent', 'Count', 'Density'))
    )
})
## Group variable? --> Just necessary for Plot_type=Scatterplot ##
output$group2 <- renderUI({
  if (is.null(data1()) || input$Plot_type != "Scatterplot") return()
  tagList(
    selectInput(inputId="group", label="Choose the grouping variable", multiple=FALSE,
                names(data1()))
  )
})
## Ratio of the boxes? --> Just necessary for Plot_type=Boxplot ##
output$group <- renderUI({
  if (is.null(data1()) || input$Plot_type != "Boxplot") return()
  tagList(
    sliderInput(inputId="box_ratio", label="Choose the ratio of the boxes", min=0, max=1,
                value=0.5, step=0.1)
  )
})
## Additional split variable? --> 1 or 2 splitting variable(s) can be chosen for each graph ##
output$split <- renderUI({
  if (is.null(data1())) return()
  tagList(
    selectInput(inputId="split", label="Choose one or two variables as splitting variable
                (optional)", multiple=TRUE, names(data1()))
  )
})


####### tabPanel 1: Create the plot #######

output$plot <- renderPlot({

  if (is.null(data1()) || input$Plot_type == "") return()

  # Histogram #
  else if (input$Plot_type == "Histogram") {
    x <- input$x
    split <- input$split
    if (is.null(split)) {
      if (input$hist_type == "Percent") {histogram( ~ data1()[,which(names(data1())==x)],
          data=data1(), xlab=x, main=input$title, type="percent")}
      else if (input$hist_type == "Count") {histogram( ~ data1()[,which(names(data1())==x)],
               data=data1(), xlab=x, main=input$title, type="count")}
      else if (input$hist_type == "Density") {histogram( ~ data1()[,which(names(data1())==x)],
               data=data1(), xlab=x, main=input$title, type="density")}
    }
    else {
      if (length(split)==1) {
        if (input$hist_type == "Percent") {histogram( ~ data1()[,which(names(data1())==x)] |
                          get(split), data=data1(), xlab=x, main=input$title, type="percent")}
        else if (input$hist_type == "Count") {histogram( ~ data1()[,which(names(data1())==x)] |
                          get(split), data=data1(), xlab=x, main=input$title, type="count")}
        else if (input$hist_type == "Density") {histogram( ~ data1()[,which(names(data1())==x)]
                         | get(split), data=data1(), xlab=x, main=input$title, type="density")}
      }
      else if (length(split)==2) {
        if (input$hist_type == "Percent") {histogram( ~ data1()[,which(names(data1())==x)] |
         get(split[1])*get(split[2]), data=data1(), xlab=x, main=input$title, type="percent")}
        else if (input$hist_type == "Count") {histogram( ~ data1()[,which(names(data1())==x)] |
            get(split[1])*get(split[2]), data=data1(), xlab=x, main=input$title, type="count")}
        else if (input$hist_type == "Density") {histogram( ~ data1()[,which(names(data1())==x)]
        | get(split[1])*get(split[2]), data=data1(), xlab=x, main=input$title, type="density")}
      }
```

```
  }
}

# Scatterplot #
else if (input$Plot_type == "Scatterplot") {
  x <- input$x
  y <- input$y
  group <- input$group
  split <- input$split
  if (is.null(split)) {
    if (group == "No groups") {
      xyplot(data1()[,which(names(data1())==y)] ~ data1()[,which(names(data1())==x)],
             data=data1(), xlab=x, ylab=y, main=input$title)
    }
    else {
      xyplot(data1()[,which(names(data1())==y)] ~ data1()[,which(names(data1())==x)],
             groups=data1()[,which(names(data1())==group)], data=data1(), xlab=x, ylab=y,
             main=input$title, auto.key=TRUE)
    }
  }
  else {
    if (length(split)==1) {
      if (group == "No groups") {
        xyplot(data1()[,which(names(data1())==y)] ~ data1()[,which(names(data1())==x)] |
               get(split), data=data1(), xlab=x, ylab=y, main=input$title)
      }
      else {
        xyplot(data1()[,which(names(data1())==y)] ~ data1()[,which(names(data1())==x)] |
               get(split), groups=data1()[,which(names(data1())==group)], data=data1(),
               xlab=x, ylab=y, main=input$title, auto.key=TRUE)
      }
    }
    else if (length(split)==2) {
      if (group == "No groups") {
        xyplot(data1()[,which(names(data1())==y)] ~ data1()[,which(names(data1())==x)] |
               get(split[1])*get(split[2]), data=data1(), xlab=x, ylab=y, main=input$title)
      }
      else{
        xyplot(data1()[,which(names(data1())==y)] ~ data1()[,which(names(data1())==x)] |
               get(split[1])*get(split[2]), groups=data1()[,which(names(data1())==group)],
               data=data1(), xlab=x, ylab=y, main=input$title, auto.key=TRUE)
      }
    }
  }
}

# Boxplot #
else if (input$Plot_type == "Boxplot") {
  x <- input$x
  y <- input$y
  split <- input$split
  if (is.null(split)) {
    bwplot(data1()[,which(names(data1())==y)] ~ data1()[,which(names(data1())==x)],
           data=data1(), box.ratio=input$box_ratio, xlab=x, ylab=y, main=input$title)
  }
  else {
    if (length(split)==1) {
      bwplot(data1()[,which(names(data1())==y)] ~ data1()[,which(names(data1())==x)] |
             get(split), data=data1(), box.ratio=input$box_ratio, xlab=x, ylab=y,
             main=input$title)
    }
    else if (length(split)==2) {
      bwplot(data1()[,which(names(data1())==y)] ~ data1()[,which(names(data1())==x)] |
             get(split[1])*get(split[2]), data=data1(), box.ratio=input$box_ratio, xlab=x,
             ylab=y, main=input$title)
    }
  }
}
```

```
  }, height=function(x) input$hei, width=function(x) input$wid) # Height and weight of the plot


  ####### tabPanel 2: Show the dataset #######
  output$dataset <- renderTable({
    data1()
  })

}


shinyApp(ui = ui, server = server)
```

CONCLUSION

R Shiny is an efficient tool to create interactive web applications without knowing any HTML, but just the programming language R. If the R programmer knows how to set up the two components of a Shiny app - the user interface and the server function - and how they affect each other, these web applications can be a helpful tool to support non-statisticians and non-programmers to explore the data on their own without knowing R.

The flexibility and the layout of the app can be chosen by the programmer. However, the more flexible the app shall be, the more complex the R code becomes and the transparency of the app can decrease hereby. Therefore, it is worth to draw up easy-handling requirements for the app (e.g. the dataset could have a specific format for the upload or the variables of the dataset should be predefined), so that the app is still useful for the investigator and also programmable for the developer of the app.

REFERENCES

RStudio (2017). *Learn Shiny*. https://shiny.rstudio.com/tutorial/ (from 30.09.2019)
SAS Institute Inc. (2017). *Sashelp Data Sets*. Cary, NC, USA.
RStudio (2014). *Shiny Themes*. https://rstudio.github.io/shinythemes/ (from 04.10.2019)

RECOMMENDED READING

RStudio (2017). *Gallery*. https://shiny.rstudio.com/gallery/ (from 04.10.2019)

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name	Stephanie Fechtner
Company	Bayer AG
Address	Aprather Weg 18a
City / Postcode	Wuppertal, 42096, Germany
Work Phone:	+49 202 36-3833
Email:	stephanie.fechtner@bayer.com
Web:	www.bayer.com