Creating Shiny Apps for biostatisticians and bioinformaticians

ISGlobal

21-22 May 2019

Isaac Subirana

Part V: Advances issues

Processing math: 100%

Outline

Part V: Advanced issues

- `observe` and `observeEvent` functions
- Updating elements
- Reactive variables
- `hide`, `show`, `toggle` and `disable` functions
- Exercise

observe & observeEvent

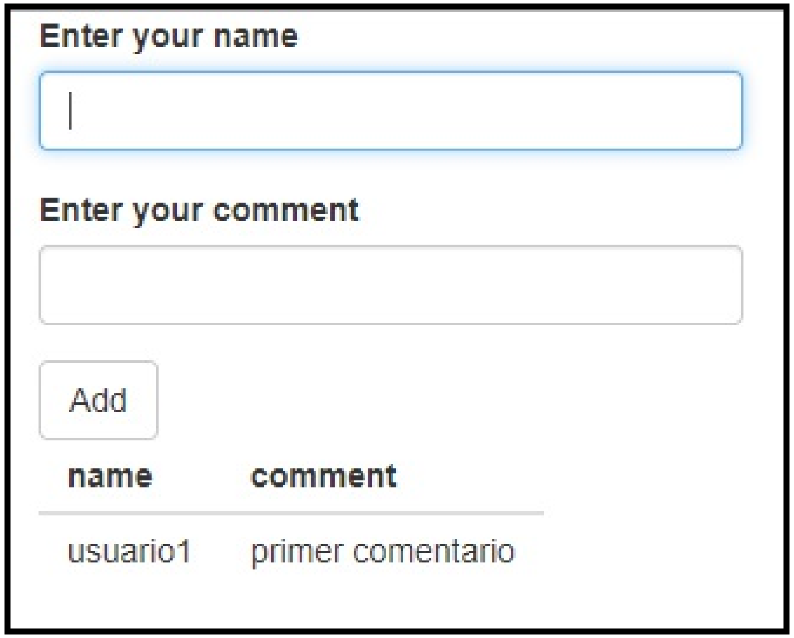
observeEvent

- The `observeEvent` function is meant to execute instructions inside the Server section when **one** element is changed/updated.
- All the code inside `observeEvent` will be exectued only if this element is changed.

```
server <- function(input, output){
  ...
   observeEvent(input$element,{
     ...
     instructions
     ...
   })
  ...
}
```

Example: Add a register

```r
if (!file.exists("commentsTable.rda")){
    commentsTable <- data.frame(name=character(),
                                comments=character())
    save(commentsTable, file="commentsTable.rda")
}

ui <- fluidPage(
    textInput("name", "Enter your name"),
    textInput("comment", "Enter your comment"),
    actionButton("add", "Add"),
    tableOutput("comments")
)

server <- function(input, output){
   observeEvent(input$add, {
     load("commentsTable.rda")
     temp <- data.frame(name = input$name,
                        comment = input$comment)
     commentsTable <- rbind(commentsTable, temp)
     save(commentsTable, file="commentsTable.rda")
   })
  output$comments <- renderTable({
    input$add
    load("commentsTable.rda")
    commentsTable
  })
}

shinyApp(ui, server)
```



name	comment
usuario1	primer comentario

- Every time the "add" button is pressed a new line is added to a database
- There must exist the data base with the variable and no rows in the server

observe

- The first argument of `observeEvent` function is a single element.
- But, what happens if you desire to execute the instructions if one out of **several** elements change?
- Then `observe` function is used instead of `observeEvent`. You can use `isolate`.

```
server <- function(input, output){
  ...
   observe({
     input$element1
     input$element2
     isolate({
       instructions
     })
   })
  ...
}
```

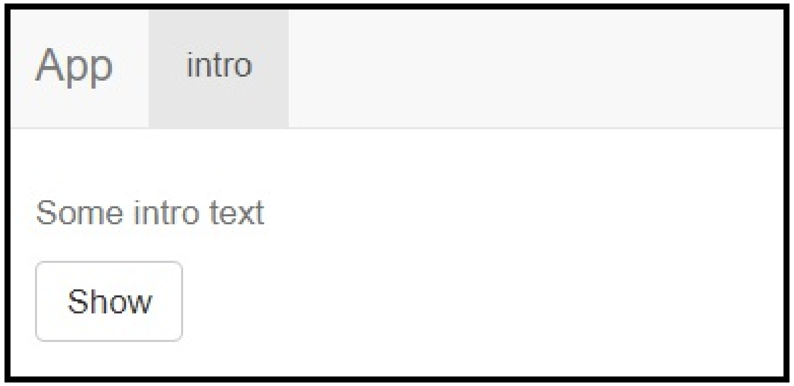
- `observe` can be used to execute some instructions when the app is launched.
- To do so, leave the function empty.

```
server <- function(input, output){
  ...
   observe({
     instructions
   })
  ...
}
```

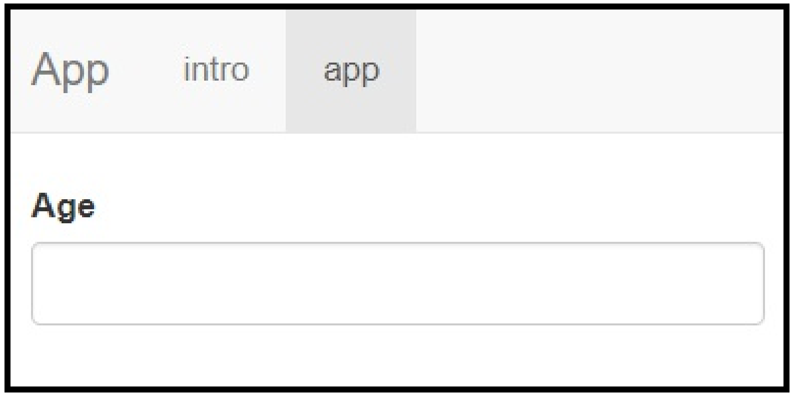
Example: Show and hide tabs

- The app has two tabs. Only the first must be shown at the begining
- When the button is pressed, the second tab is shown.

```r
library(shiny)

ui <- navbarPage(title="App", id="menu",
  tabPanel("intro",
    helpText("Some intro text"),
    actionButton("show", "Show")
  ),
  tabPanel("app",
    numericInput("age","Age",NA)
  )
)

server <- function(input, output){
  observe({
    hideTab(inputId="menu", target="app")
  })
  observeEvent(input$show,{
    showTab(inputId="menu", target="app")
  })
}

shinyApp(ui, server)
```



Some intro text



Updating elements

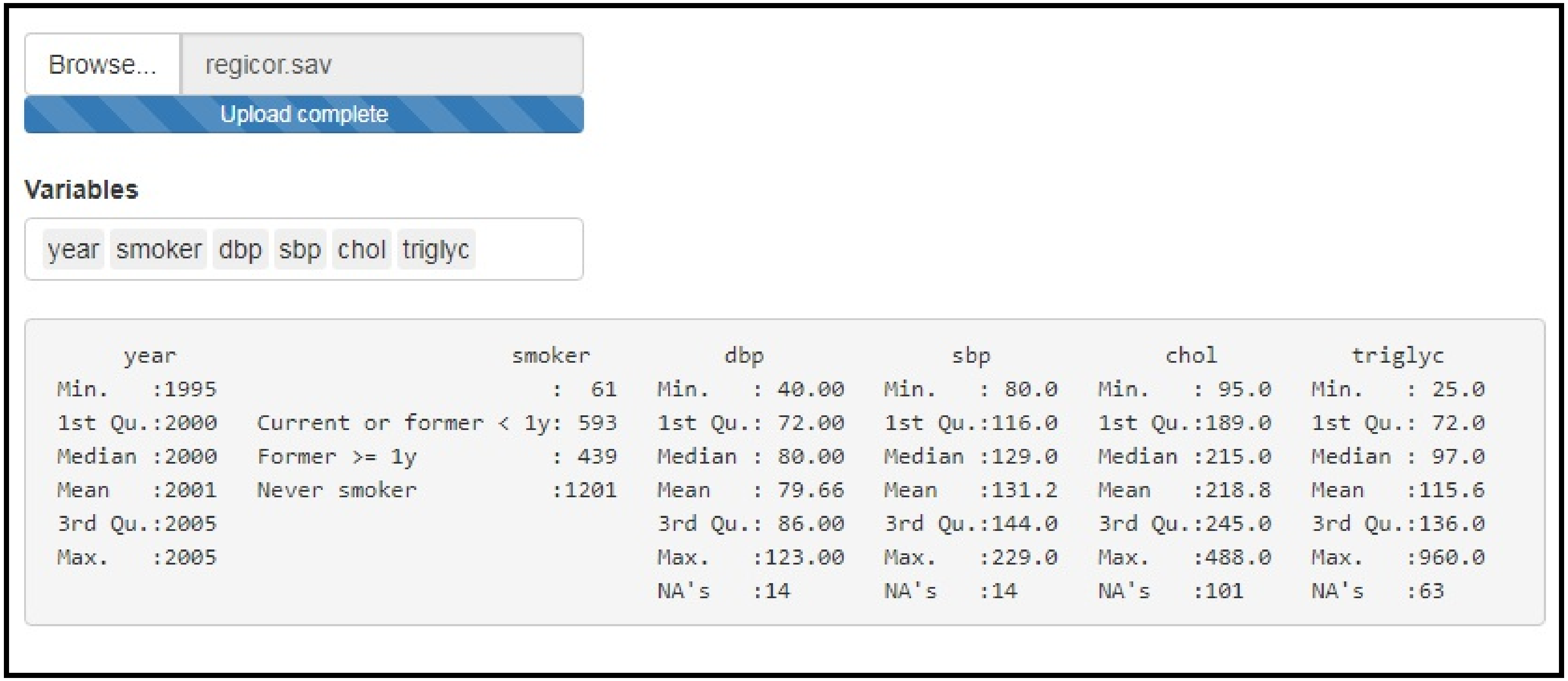
Updating elements

- Unlike `uiOutput / renderUI`, using the **`update***`** functions only those specified arguments are modified and the others remain as they are.
- **`update***`** functions are used inside Server section.
- Specifically, they are called inside `observe` or `observeEvent`.
- Note the argument **`session`** when defining Server function.

Initialization	Update	Modifiable arguments
textInput	updateTextInput	label, value
numericInput	updateNumericInput	label, value
checkboxInput	updateCheckboxInput	label, value
radioButtons	updateRadioButtons	label, choices, selected, inline
selectInput	updateSelectInput	label, choices, selected
sliderInput	updateSliderInput	label, value, min, max, step
actionButton	updateActionButton	label, icon
bsButton {shinyBS}	updateButton	label, icon, style, disabled

Example: variable list

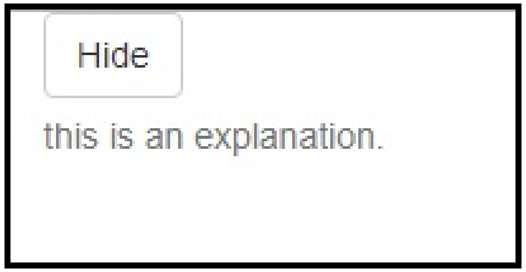
```r
library(haven)

ui <- fluidPage(
  fileInput("file", ""),
  selectInput("vars", "Variables", choices=NULL, multiple=TRUE),
  verbatimTextOutput("summary")
)

server <- function(input, output, session){

  dat <- reactive({
    if (is.null(input$file)) return(invisible(NULL))
    haven::read_sav(input$file$datapath)
  })

  output$summary <- renderPrint({
    if (length(input$vars)==0) return(NULL)
    summary(dat()[,input$vars])
  })

   observe({
     updateSelectInput(session, "vars", choices = names(dat()))
   })

}

shinyApp(ui, server)
```



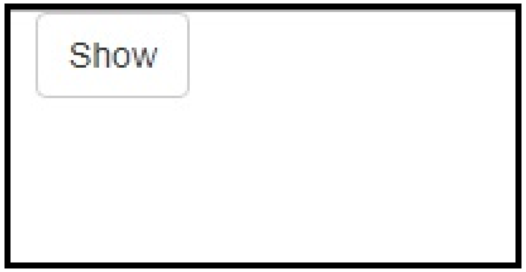
```
     year                            smoker          dbp              sbp             chol            triglyc     
 Min.   :1995                           :  61   Min.   : 40.00   Min.   : 80.0   Min.   : 95.0   Min.   : 25.0  
 1st Qu.:2000   Current or former < 1y: 593   1st Qu.: 72.00   1st Qu.:116.0   1st Qu.:189.0   1st Qu.: 72.0  
 Median :2000   Former >= 1y            : 439   Median : 80.00   Median :129.0   Median :215.0   Median : 97.0  
 Mean   :2001   Never smoker            :1201   Mean   : 79.66   Mean   :131.2   Mean   :218.8   Mean   :115.6  
 3rd Qu.:2005                                   3rd Qu.: 86.00   3rd Qu.:144.0   3rd Qu.:245.0   3rd Qu.:136.0  
 Max.   :2005                                   Max.   :123.00   Max.   :229.0   Max.   :488.0   Max.   :960.0  
                                                NA's   :14       NA's   :14      NA's   :101     NA's   :63     
```

Example 2: Button (label)

```r
ui <- fluidPage(
  actionButton("help", "Hide", width = "60px"),
  conditionalPanel(
    condition = "input.help%2==0",
    helpText("this is an explanation.")
  )
)

server <- function(input, output, session){

  observeEvent(input$help, {
    if (input$help%%2==0)
      updateActionButton(session, "help", label="Hide")
    else {
      updateActionButton(session, "help", label="Show")
    }
  })

}

shinyApp(ui, server)
```

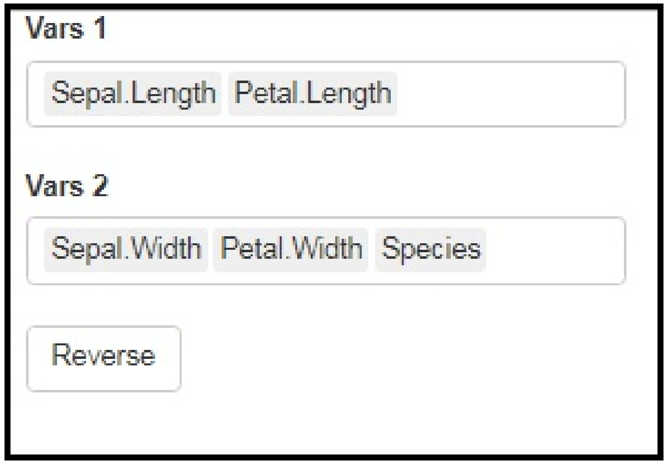


this is an explanation.



Example 3: switch selected items

```r
library(shiny)
vars <- names(iris)

ui <- fluidPage(
  selectInput("vars1", "Vars 1", choices=vars, multiple=TRUE),
  selectInput("vars2", "Vars 2", choices=vars, multiple=TRUE),
  actionButton("switch","Switch")
)

server <- function(input, output, session){

  observeEvent(input$vars1,{
     updateSelectInput(session, "vars2",
                       choices=vars[!vars%in%input$vars1],
                       selected=input$vars2)
  })

  observeEvent(input$switch,{
    vars1 <- input$vars1
    vars2 <- input$vars2
     updateSelectInput(session, "vars1",
                       choices=vars, selected=vars2)
     updateSelectInput(session, "vars2",
                       choices=vars, selected=vars1)
  })

}

shinyApp(ui, server)
```

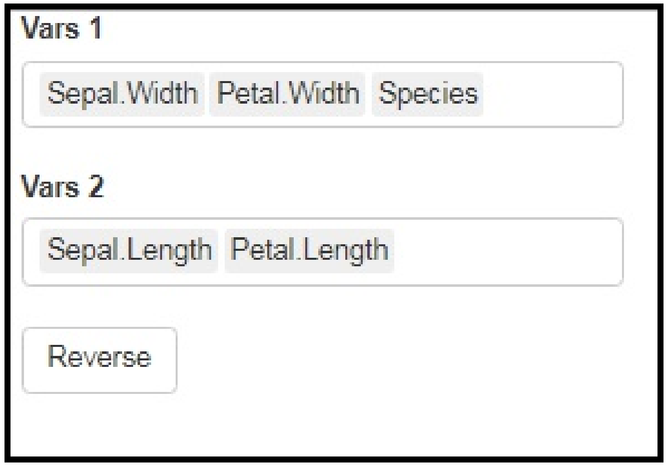

Vars 1

Sepal.Length Petal.Length

Vars 2

Sepal.Width Petal.Width Species

Reverse


Vars 1

Sepal.Width Petal.Width Species

Vars 2

Sepal.Length Petal.Length

Reverse

Note: This cannot be done by `renderUI` and `uiOutput`.

Reactive variables

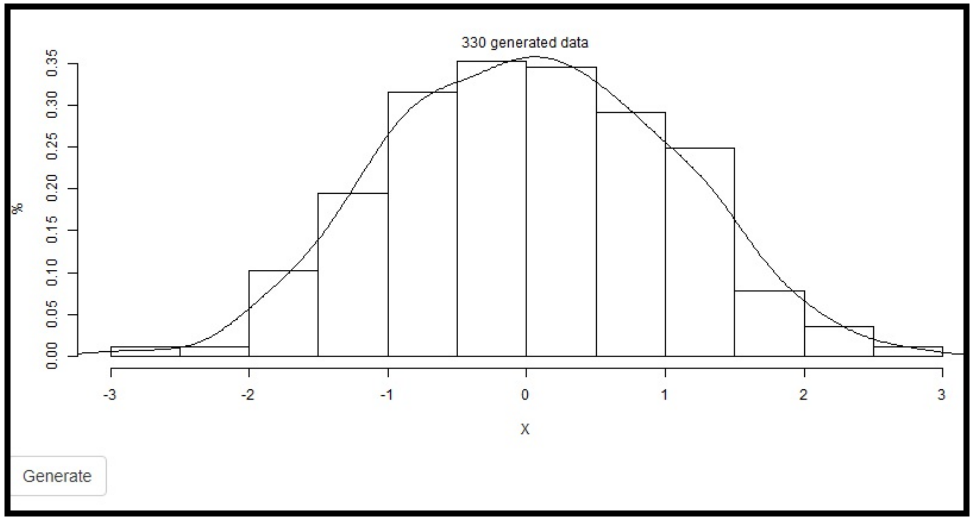
Reactive variables

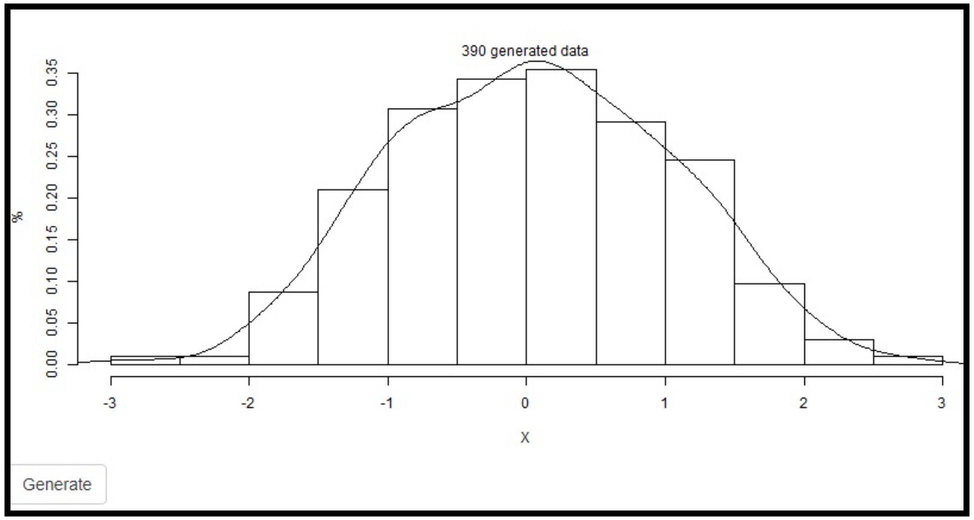
- **Reactive variables** are objects whose values are modified in a "reactive" way
- They exists inside the server section.
- Every *reactive values* type object is an element of a list which is iniciated using `reactiveValues` function.

```
server <- function(input, output){

  rv <- reactiveValues()
  rv$element <- 0
  ...

}
```

Example 1: Cumulating data plot

- Generate data from a normal distribution when pressing a button.
- The generated data must be added to already generated ones.
- Plot an histogram with all the cumulated data.



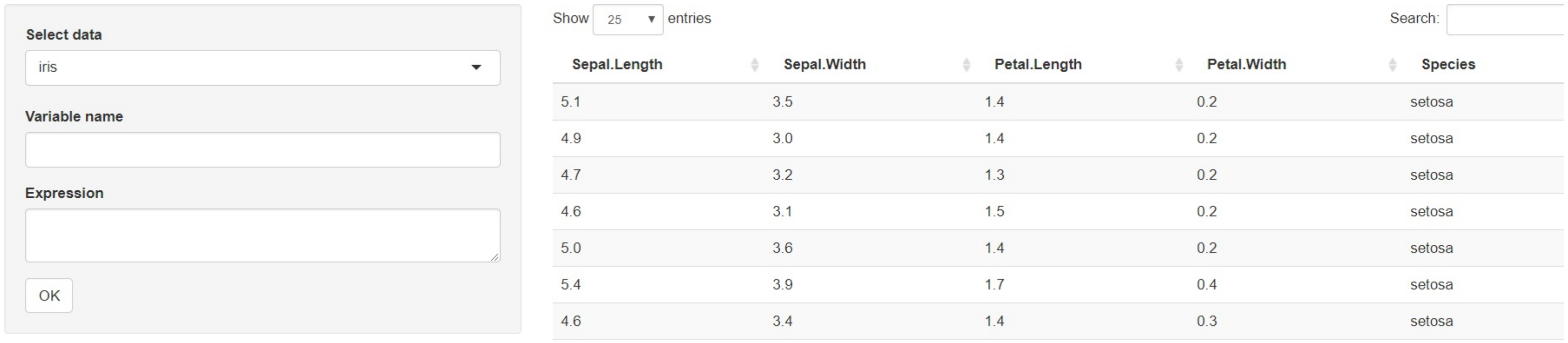


```r
ui<-bootstrapPage(
  plotOutput("plot"),
  actionButton("go", "Generate")
)

server<-function(input,output){

  rv <- reactiveValues()
  rv$numbers <- numeric()

  observeEvent(input$go,{
    rv$numbers <- c(rv$numbers,rnorm(10))
  })

  output$plot <- renderPlot({
    if (length(rv$numbers)==0) # no data yet
      return(invisible(NULL))
    hist(rv$numbers, freq=FALSE, xlab="X", ylab="%",main="")
    lines(density(rv$numbers))
    mtext(paste(length(rv$numbers), "generated data"))
  })
}

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

Note that `renderPlot` is executed when `rv$numbers` change.

Example 2: Recode/create variables

```r
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(
      selectInput("dataname", "Select data", c("iris","mtcars")),
      textInput("varname","Variable name"),
      textAreaInput("expr","Expression"),
      actionButton("ok","OK")
    ),
    mainPanel(
      dataTableOutput("datatable")
    )
  )
)

server <- function(input, output){
  rv <- reactiveValues(data=data.frame)
   observeEvent(input$dataname,{
     rv$data <- get(input$dataname)
   })
   observeEvent(input$ok,{
     rv$data[,input$varname] <- with(rv$data, eval(parse(text=input$expr)))
   })
  output$datatable <- renderDataTable(rv$data)
}

shinyApp(ui, server)
```



Select data

iris

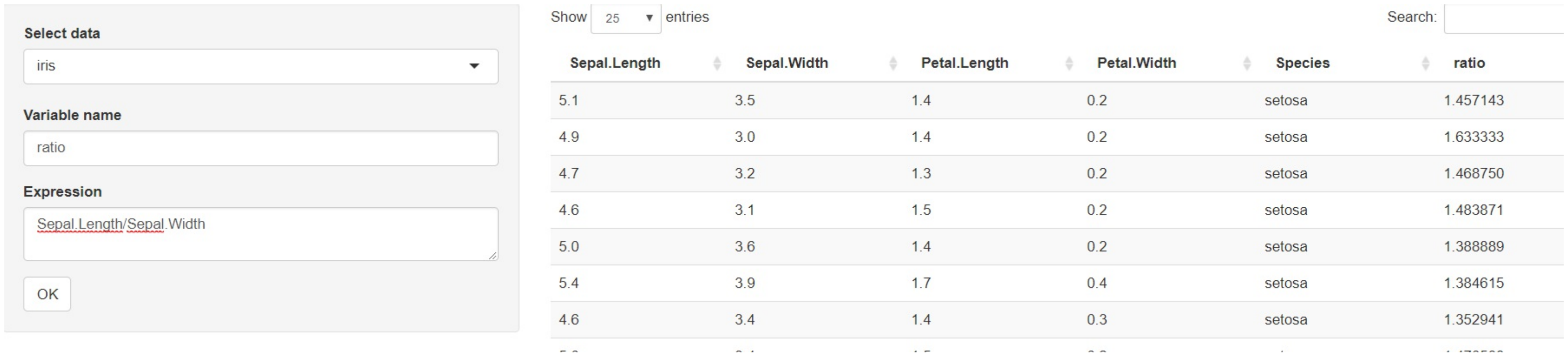
Variable name

Expression

OK

Show 25 entries   Search:

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa



Select data

iris

Variable name

ratio

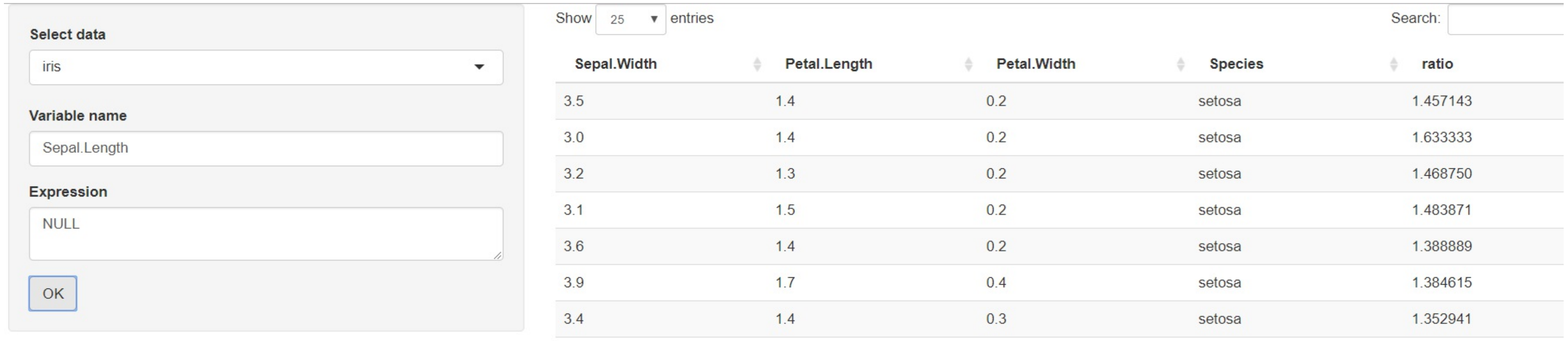
Expression

Sepal.Length/Sepal.Width

OK

Show 25 entries   Search:

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	ratio
5.1	3.5	1.4	0.2	setosa	1.457143
4.9	3.0	1.4	0.2	setosa	1.633333
4.7	3.2	1.3	0.2	setosa	1.468750
4.6	3.1	1.5	0.2	setosa	1.483871
5.0	3.6	1.4	0.2	setosa	1.388889
5.4	3.9	1.7	0.4	setosa	1.384615
4.6	3.4	1.4	0.3	setosa	1.352941



Select data

iris

Variable name

Sepal.Length

Expression

NULL

OK

Show 25 entries   Search:

Sepal.Width	Petal.Length	Petal.Width	Species	ratio
3.5	1.4	0.2	setosa	1.457143
3.0	1.4	0.2	setosa	1.633333
3.2	1.3	0.2	setosa	1.468750
3.1	1.5	0.2	setosa	1.483871
3.6	1.4	0.2	setosa	1.388889
3.9	1.7	0.4	setosa	1.384615
3.4	1.4	0.3	setosa	1.352941

Toggle, show, hide & disable

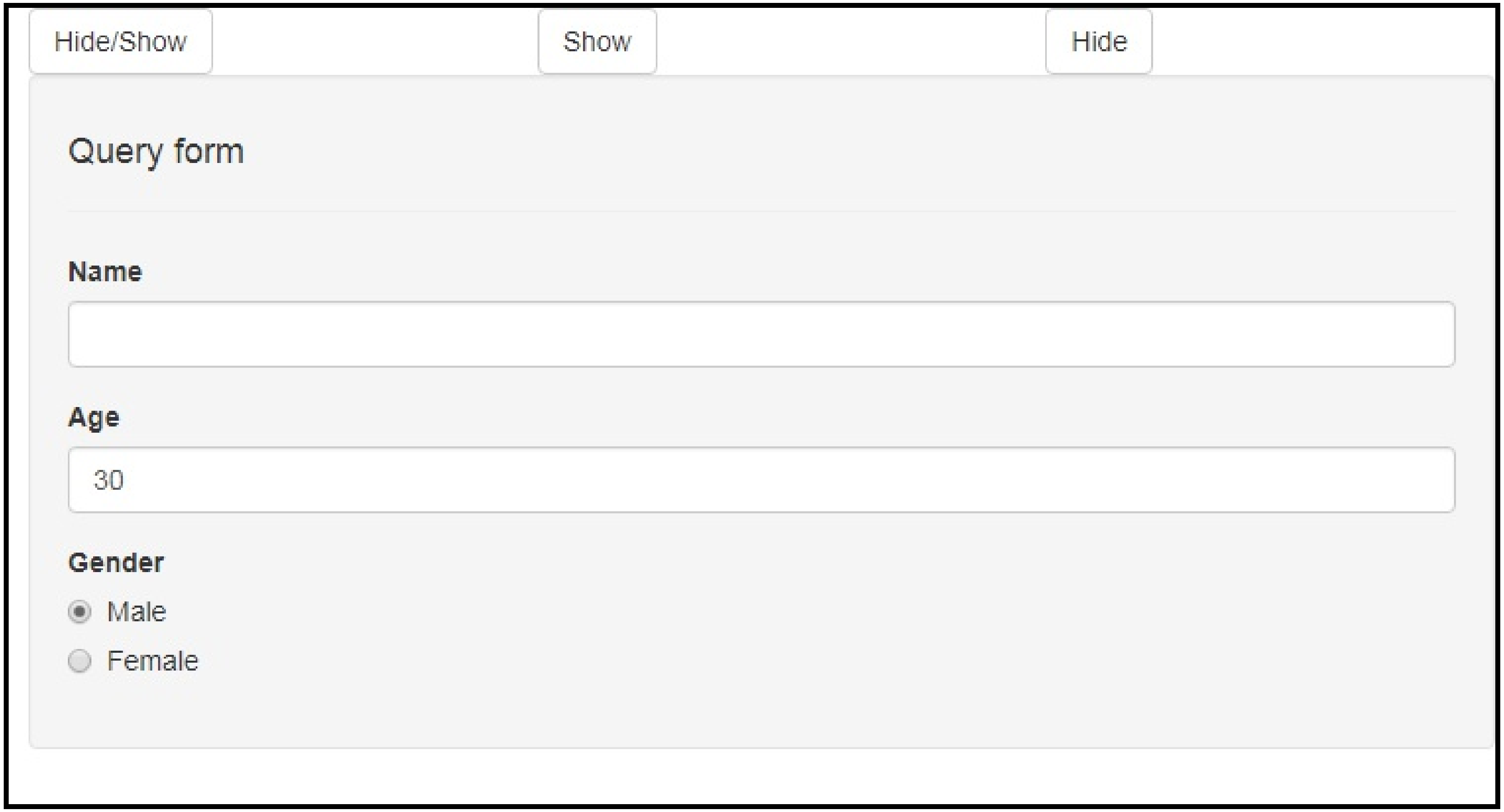
Toggle, show, hide & disable

- The `shinyjs` package, among other features, allows:
  - Hide/Show form widgets (`hide`, `show`, `toggle`)
  - Enable or disable buttons or other input widgets (`disable`)
- It is available on CRAN:

```
install.packages(shinyjs)
```

- For more info, visit its website.

Example 1. Buttons (hide, show, toggle)



Query form

Name

Age

30

Gender

(●) Male

( ) Female

UI

```r
ui <- fluidPage(

  useShinyjs(), # Set up shinyjs

  fluidRow(
    column(4,
      actionButton("btntoggle", "Hide/Show")
    ),
    column(4,
      actionButton("btnshow", "Show")
    ),
    column(4,
      actionButton("btnhide", "Hide")
    )
  ),

  hidden(
    wellPanel(id="elem",
      h4("Query form"),
      hr(),
      textInput("name", "Name", ""),
      numericInput("age", "Age", 30),
      radioButtons("gender", "Gender",
                   c("Male", "Female"))
    )
  )
)
```

Server

```r
server <- function(input, output) {

  observeEvent(input$btntoggle, {
    shinyjs::toggle("elem", TRUE, "fade")
  })

  observeEvent(input$btnshow, {
    shinyjs::show("elem", TRUE, "slide")
  })

  observeEvent(input$btnhide, {
    shinyjs::hide("elem", FALSE)
  })

}
```

Example 2. Password

1. Make the app visible only if the password is correct.
2. Once the correct password is introduced the password input widget and the check button must disapear

```
ui <- fluidPage(
  useShinyjs(),
  # password panel
  div(id="passScreen",
    passwordInput("pass","Password",""),
    actionButton("check","Check")
  ),
  # app
  shinyjs::hidden(
    div(id="myapp",
      titlePanel("Hello Shiny!"),
      sidebarLayout(
        sidebarPanel(
          sliderInput("obs","Number obs.",
              min=1,max=1000,value=500)
        ),
        mainPanel(
          plotOutput("distPlot")
        )
      )
    )
  )
)
```

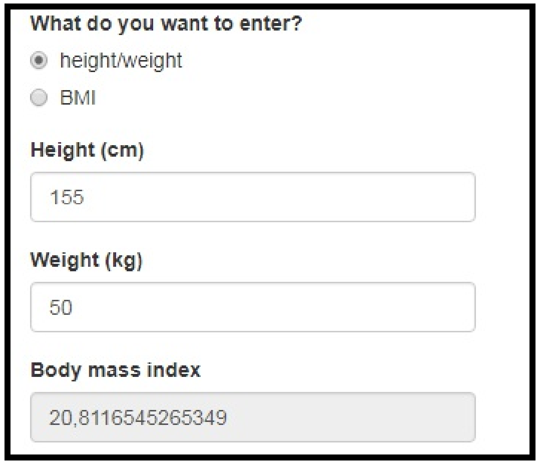
```
server <- function(input, output) {

  observeEvent(input$check, {
     if (input$pass=='123'){
       shinyjs::show("myapp", FALSE)
       shinyjs::hide("passScreen", FALSE)
     }
  })

  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })

}
```

Try it here

Example 3. Body mass index

- You can enter either weight and height or body mass index (bmi) , $weight/height^2$.
- If user enter height and weight, bmi input widgets must be disabled but visible and updated according to bmi.
- If user enter bmi, height and weight input widgets must be hidden.

```r
library(shinyjs)

ui <- fluidPage(
  useShinyjs(),
  radioButtons("what","What do you want to enter?",
               c("height/weight"=1, "BMI"=2)),
  numericInput("height","Height (cm)",NA),
  numericInput("weight","Weight (kg)",NA),
  numericInput("bmi","Body mass index",NA)
)

server <- function(input, output, session) {
  observe({
    if (input$what==1){
       shinyjs::disable("bmi", FALSE)
       updateNumericInput(session, "bmi",
           value=input$weight/(input$height/100)^2)
       shinyjs::show("height", FALSE)
       shinyjs::show("weight", FALSE)
    } else {
       shinyjs::enable("bmi", FALSE)
       shinyjs::hide("height", FALSE)
       shinyjs::hide("weight", FALSE)
    }
  })
}

shinyApp(ui, server)
```



What do you want to enter?

[✓ height/weight]

BMI

Height (cm)

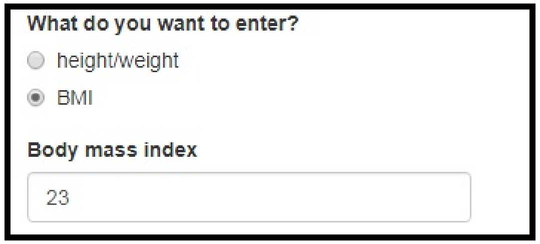
155

Weight (kg)

50

Body mass index

20,8116545265349



What do you want to enter?

height/weight

[✓ BMI]

Body mass index

23

Practice

Create your application

Proposal 1:

From the app created in part IV, add a password that be hidden once the pass is correct (123)

Try it here

Proposal 2:

Create an app to perform a simple regression from iris data

- Left panel:
  - Ask for response variable
  - Ask for independent variable (response must be removed from the choices)
- Right panel:
  - First tab with the summary of the regression
  - Second tab with the data plot and regression line

```
summary(mod <- lm(y ~ x))
```

```
plot(x, y)
abline(mod)
```