Tutorial in R Shiny package: Developing Web Applications in the area of Biostatistics & Data Science

Martial Luyts **Jeroen Sichien**

Interuniversity Institute for Biostatistics and statistical Bioinformatics (I-BioStat)

Katholieke Universiteit Leuven, Belgium

martial.luyts@kuleuven.be & jeroen.sichien@kuleuven.be

www.ibiostat.be

Belgium, 30 March 2016

Contents

1. Introductory material . . . 2
   - 1.1. Motivation . . . 3
   - 1.2. Introduction to R Shiny . . . 5
   - 1.3. Example . . . 10
2. Interface development . . . 12
   - 2.1. Inputs within the UI framework . . . 13
   - 2.2. Outputs within the UI framework . . . 15
   - 2.3. Assemble UI framework . . . 17

2.4. Layout structure of the UI 21

3. Learn to build an app in Shiny 25

3.1. Step-by-step approach 26

3.2. Focus on special reactive functions 49

3.3. Progress dynamic user interface 63

3.4. Extension to dashboard shells 80

4. Apps in the area of biostatistics & data science 109

4.1. Shiny as an extra tool 110

4.2. Other way of constructing Shiny apps 111

4.3. To the applications 112

References 113

Part 1:

Introduction, motivation and example

Chapter 1: Introductory material

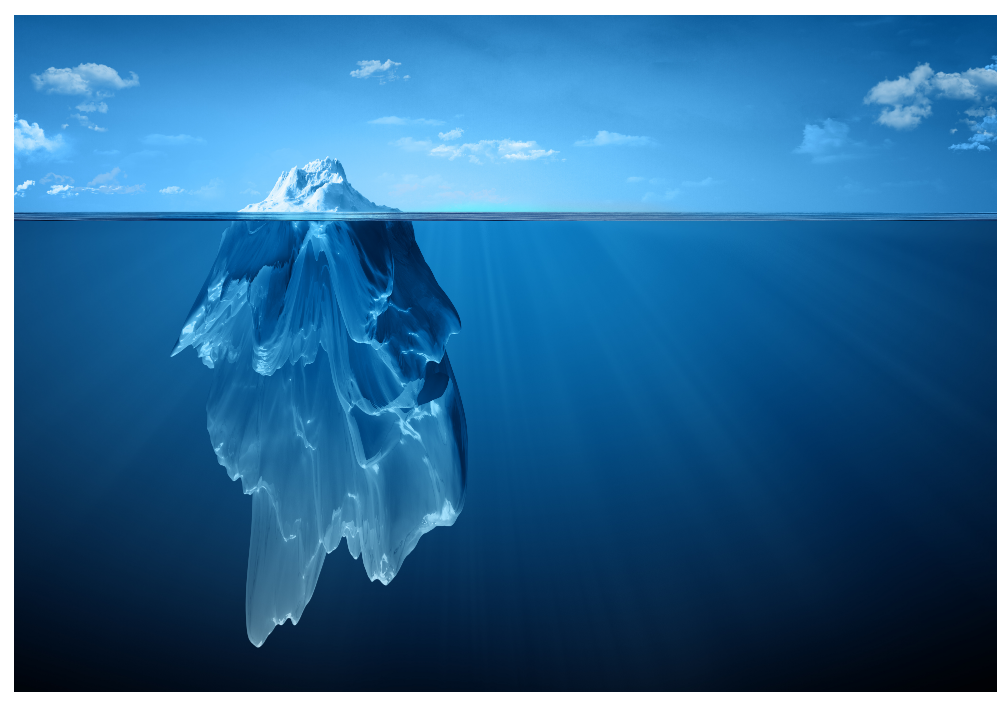
- Motivation
- Introduction to R Shiny
- Example

1.1 Motivation

- Biostatisticians often employ their analysis in **R**

- Presenting/sharing their results are often done in a static format ...

- **Problem:** Additionally questions related to their work cannot be showed immediately



- **Simple idea:** Move the water surface downwards to give more knowledge to the audience (e.g. medical doctors)

- Possible solution for gaining this information: **R Shiny**

1.2 Introduction to R Shiny

- R Shiny = R + interactivity + web made easy
  In words: Open source R package from Rstudio that creates interactive web applications around your R analyses and visualizations

- No HTML/CSS/Javascript knowledge required to implement ...

- .... but fully customizable and extensible with HTML/CSS/JavaScript

- Deploying Shiny Apps can happen in different ways:
  - **Shiny Server (AGPLv3)**

    https://github.com/rstudio/shiny-server

  - **Shiny Server "Pro"**

    Secure user access, tune application performance, monitor resource utilization and get the direct support from R Studio, Inc.

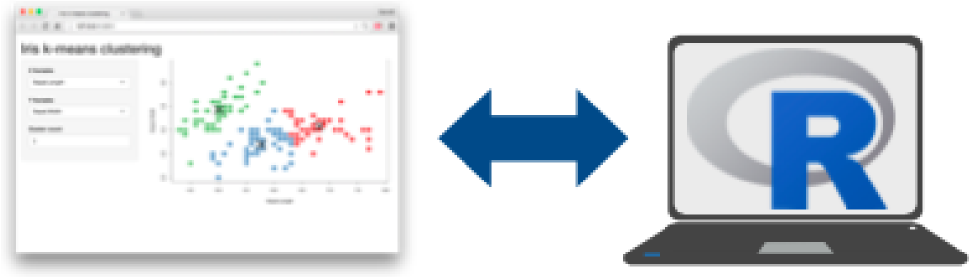
    https://www.rstudio.com/pricing/

  - **Shiny hosting on rstudio.com**

    http://www.shinyapps.io

- **Remark:** Basic knowledge of R code required

- **What's a Shiny App?**
  A **Shiny app** is a web page (**UI**) connected to a computer/server running a live R session (**Server**)



- Users can manipulate the UI, which will cause the server to update the UIs displays (by running R code)

- Shiny Apps can be developed with the following template in R:

app.R:

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

  - **ui**: Nested R functions that assemble an HTML user interface for the app
  - **server**: A function with instructions on how to build and rebuild the R objects displayed in the UI
  - **shinyApp**: Combines ui and server into a functioning app

- Save the template as **app.R**

- Alternatively, split template into two files named **ui.R** and **server.R**:

ui.R:

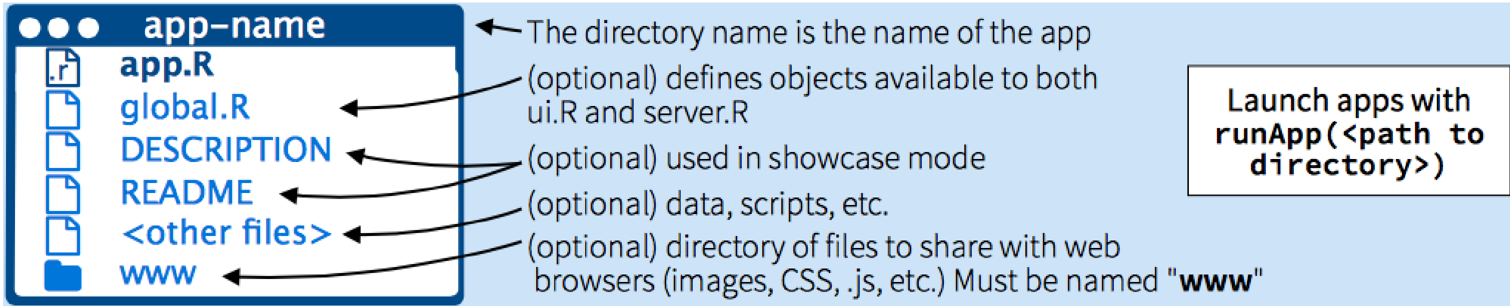
```
> fluidPage()
```

server.R:

```
> function(input, output){}
```

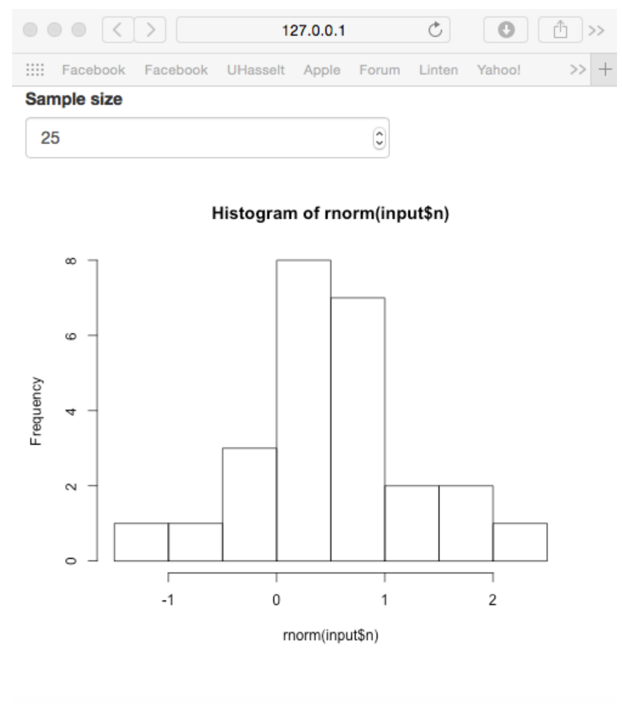
  - **Remark:** No need to call **shinyApp()**

- Save each app as a directory that contains an **app.R** file (or a **ui.R** file and a **server.R** file) plus optional extra files



1.3 Example

```
> library(shiny)
> ui <- fluidPage(
         numericInput(inputId = "n",
           "Sample size", value = 25),
         plotOutput(outputId = "hist"))
> server <- function(input, output){
         output$hist <- renderPlot({
          hist(rnorm(input$n))
         })}
> shinyApp(ui = ui, server = server)
```

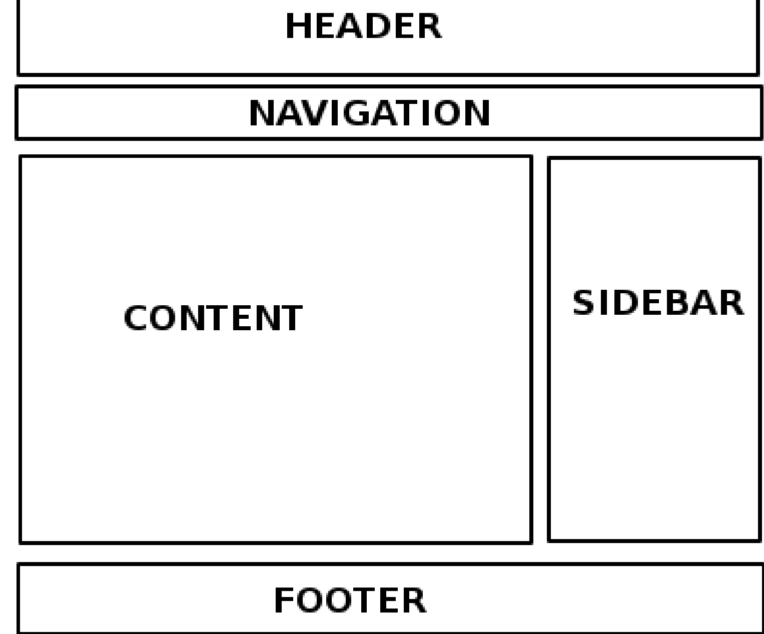


Part 2:

Layouts and functions

Chapter 2:
Interface development

- Design & explore UI framework
  - Inputs within the UI framework
  - Outputs within the UI framework
- Assemble UI with HTML/CSS/... widgets
- Adjustment of the layout scheme

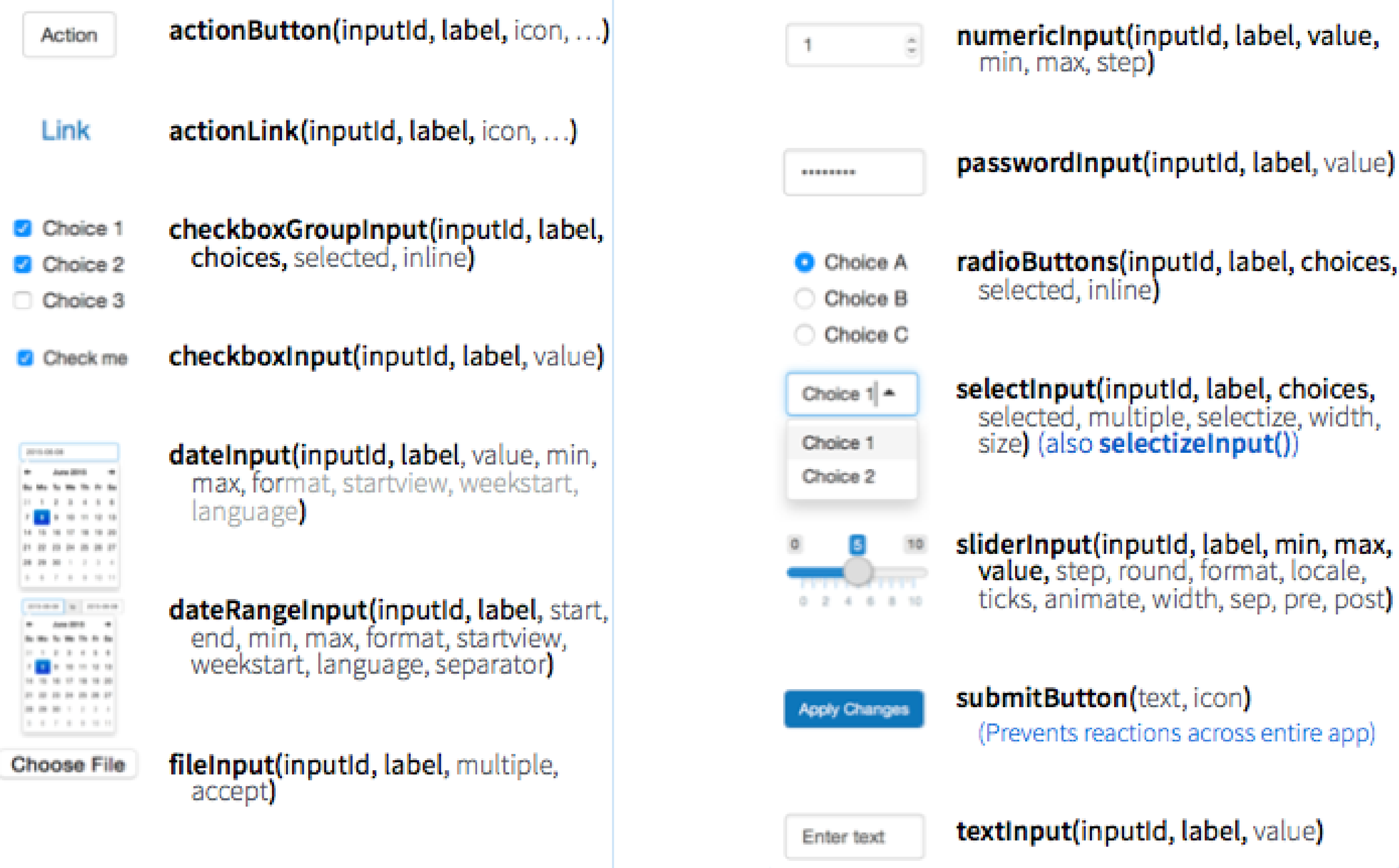


2.1 Inputs within the UI framework

```
> library(shiny)
> ui <- fluidPage(
          numericInput(inputId = "n",
           "Sample size", value = 25),
          plotOutput(outputId = "hist"))
> server <- function(input, output){
          output$hist <- renderPlot({
           hist(rnorm(input$n))
          })}
> shinyApp(ui = ui, server = server)
```

- Input values are reactive
- Access the current value of an input object with $input\$ < inputId >$

- Different input functions are available:

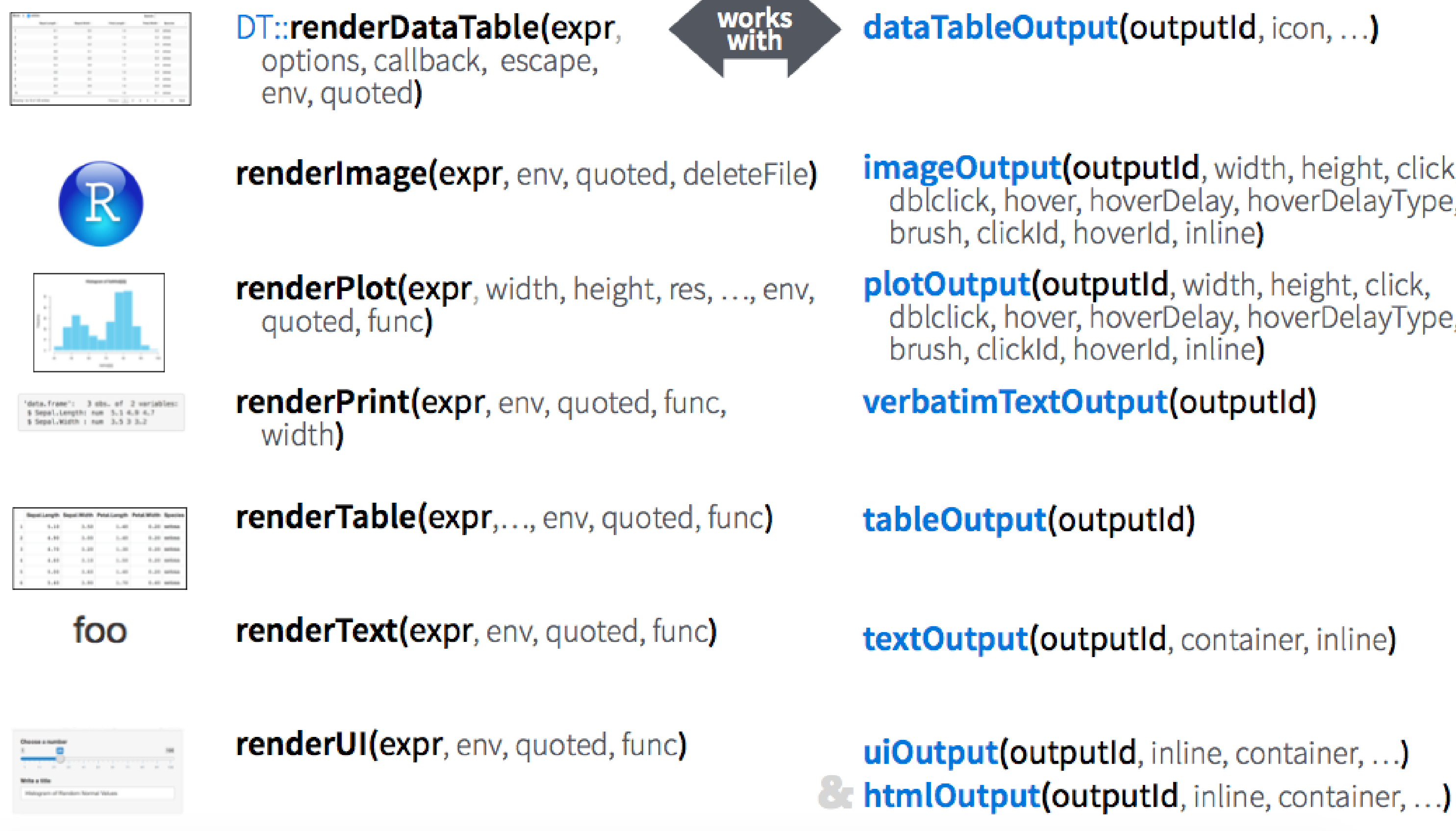


2.2 Outputs within the UI framework

```
> library(shiny)
> ui <- fluidPage(
          numericInput(inputId = "n",
            "Sample size", value = 25),
          plotOutput(outputId = "hist"))
> server <- function(input, output){
          output$hist <- renderPlot({
           hist(rnorm(input$n))
          })}
> shinyApp(ui = ui, server = server)
```

- Used to add R output to the UI framework
- Access the developed output of an output object with $output\$ < outputId >$
- render*() and *Output() functions work together

- Different output functions are available:



2.3 Assemble UI framework

- An app's UI is actually an HTML document

  ui.R:

```
> fluidPage(textInput("a",""))

## <div class="container-fluid">
##  <div class="form-group shiny-input-container">
##      <label for="a"></label>
##      <input id="a" type="text"
##          class="form-control" value=""/>
## </div>
```

- **Static HTML elements** can be added with **tags**, a list of functions that parallel common HTML tags, e.g. tags$a()

tags$a	tags$data	tags$h6	tags$nav	tags$span
tags$abbr	tags$datalist	tags$head	tags$noscript	tags$strong
tags$address	tags$dd	tags$header	tags$object	tags$style
tags$area	tags$del	tags$hgroup	tags$ol	tags$sub
tags$article	tags$details	tags$hr	tags$optgroup	tags$summary
tags$aside	tags$dfn	tags$HTML	tags$option	tags$sup
tags$audio	tags$div	tags$i	tags$output	tags$table
tags$b	tags$dl	tags$iframe	tags$p	tags$tbody
tags$base	tags$dt	tags$img	tags$param	tags$td
tags$bdi	tags$em	tags$input	tags$pre	tags$textarea
tags$bdo	tags$embed	tags$ins	tags$progress	tags$tfoot
tags$blockquote	tags$eventsource	tags$kbd	tags$q	tags$th
tags$body	tags$fieldset	tags$keygen	tags$ruby	tags$thead
tags$br	tags$figcaption	tags$label	tags$rp	tags$time
tags$button	tags$figure	tags$legend	tags$rt	tags$title
tags$canvas	tags$footer	tags$li	tags$s	tags$tr
tags$caption	tags$form	tags$link	tags$samp	tags$track
tags$cite	tags$h1	tags$mark	tags$script	tags$u
tags$code	tags$h2	tags$map	tags$section	tags$ul
tags$col	tags$h3	tags$menu	tags$select	tags$var
tags$colgroup	tags$h4	tags$meta	tags$small	tags$video
tags$command	tags$h5	tags$meter	tags$source	tags$wbr

- **Example:**

```
> ui <- fluidPage(
          tags$h1("L-Biostat App"),
          tags$hr(),
          tags$br(),
          tags$p(strong("Martial Luyts & Jeroen Sichien")),
          tags$p(em("PhD students in statistics")),
          tags$a(href="https://ibiostat.be", "I-Biostat"))
> server <- function(input, output){}
> shinyApp(ui = ui, server = server)
```

127.0.0.1

Facebook Facebook UHasselt Apple Forum Linten Yahoo!

L-Biostat App

Martial Luyts & Jeroen Sichien

PhD students in statistics

I-Biostat

- Several files can be included as well:
  - **CSS file**
    1. Place the file in the **www** subdirectory
    2. Link to it with:

```
tags$head(tags$link(rel = "stylesheet",
          type = "text/css", href = "<filename>"))
```

  - **Javascript file**
    1. Place the file in the **www** subdirectory
    2. Link to it with:

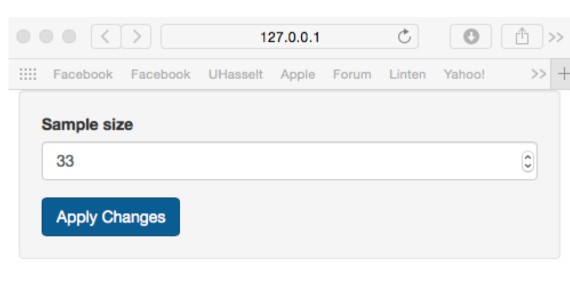
```
tags$head(tags$script(src = "<filename>"))
```

  - **Image**
    1. Place the file in the **www** subdirectory
    2. Link to it with: $img(src = " < filename > ")$

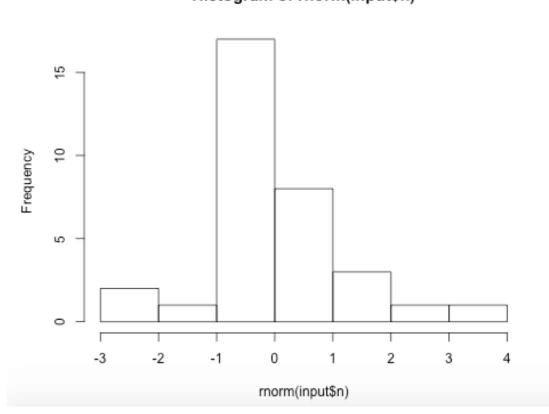
2.4 Layout structure of the UI

- Combine multiple elements into a "single element" that has its own properties with a **panel** function

Example:




```
> ui <- fluidPage(
          wellPanel(
           numericInput(inputId = "n",
            "Sample size", value = 25),
           submitButton()),
          plotOutput(outputId = "hist"))
> server <- function(input, output){...}
> shinyApp(ui = ui, server = server)
```



- Different panels are available:

absolutePanel()	inputPanel()	tabPanel()
conditionalPanel()	mainPanel()	tabsetPanel()
fixedPanel()	navlistPanel()	titlePanel()
headerPanel()	sidebarPanel()	wellPanel()

- Organize panels and elements into a layout with a **layout** function

fluidRow()

column row col

column

```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```

flowLayout()

object 1 object 2 object 3

object 3

```
ui <- fluidPage(
  flowLayout( # object 1,
              # object 2,
              # object 3
  )
)
```

sidebarLayout()

side panel main panel

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

splitLayout()

object 1 object 2

```
ui <- fluidPage(
  splitLayout( # object 1,
               # object 2
  )
)
```

verticalLayout()

object 1

object 2

object 3

```
ui <- fluidPage(
  verticalLayout( # object 1,
                  # object 2,
                  # object 3
  )
)
```

- Layer tabPanels on top of each other, and navigate between them

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

ui <- fluidPage( navlistPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")))

ui <- navbarPage(title = "Page",
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents"))
```

tab 1 tab 2 tab 3
contents

tab 1
tab 2
tab 3
contents

Page tab 1 tab 2 tab 3
contents

Part 3:

Basic Tutorial to R Shiny

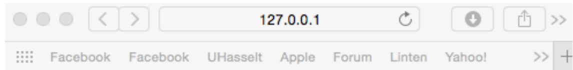
Chapter 3:
Learn to build an app in Shiny

- Step-by-step approach
- Focus on special reactive functions
- Progress dynamic user interface
- Extension to dashboard shells

3.1 Step-by-step approach

- **App 1:**
  Create a **simple app** that displays **text** within the **title panel**, **sidebar panel** and **main panel**, where the sidebar panel is located on the right

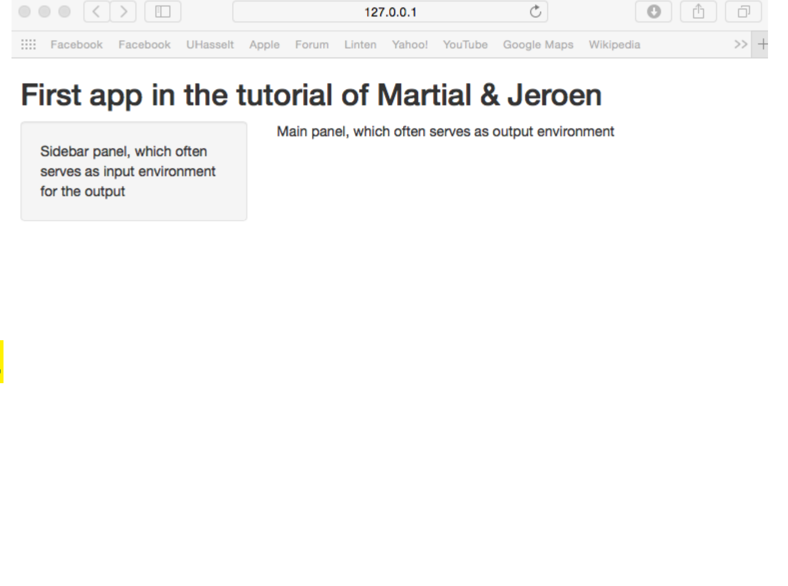
  Step 1: Install package & build framework




```
> install.packages("shiny")
> library(shiny)
> ui <- fluidPage()
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```

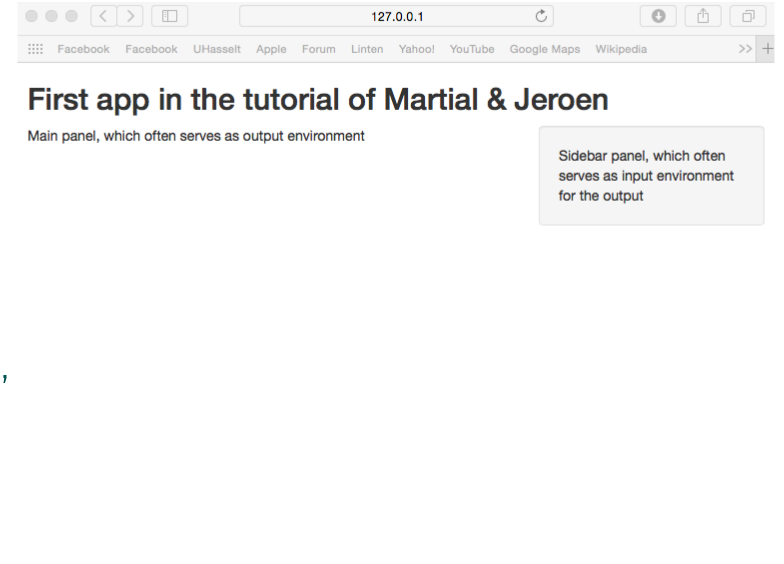
Step 2: Building the UI framework

```
> install.packages("shiny")
> library(shiny)
> ui <- fluidPage(
         titlePanel(title="First app ..."),
         sidebarLayout(
         sidebarPanel("Sidebar panel, ..."),
         mainPanel("Main panel, ..."))
         ))
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```



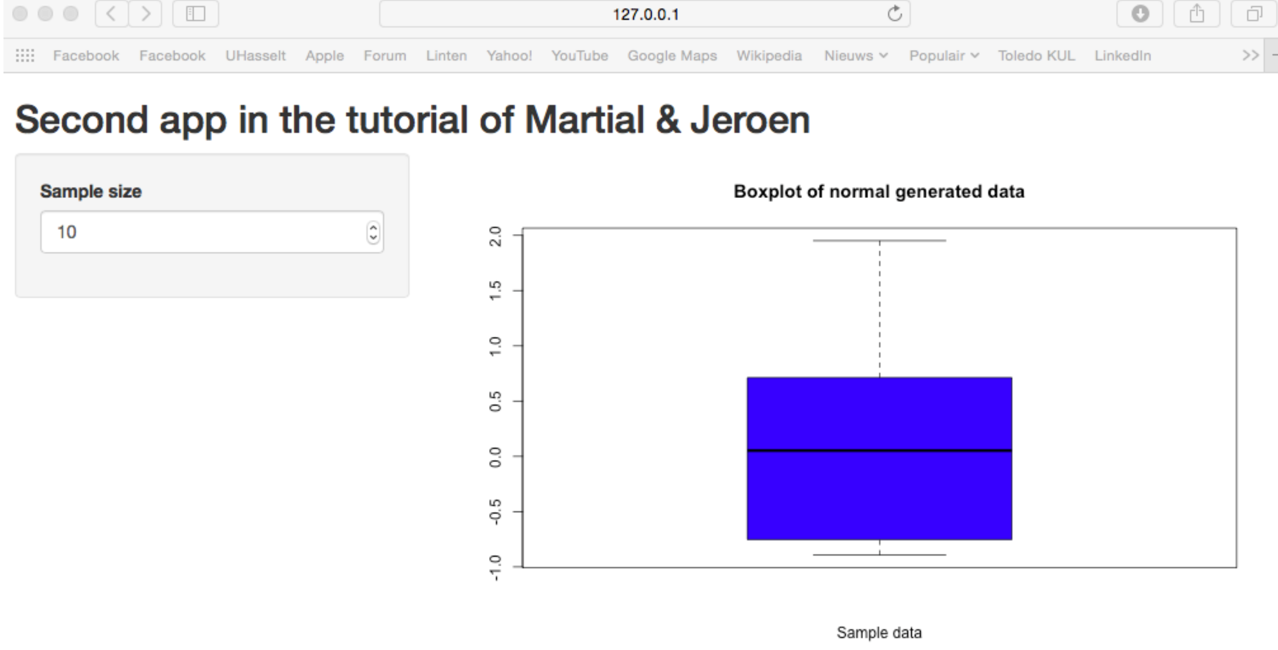
Step 3: Adjusting the UI framework

```
> install.packages("shiny")
> library(shiny)
> ui <- fluidPage(
        titlePanel(title="First app ..."),
        sidebarLayout(position="right",
        sidebarPanel("Sidebar panel, ..."),
        mainPanel("Main panel, ..."))
        ))
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```



- **App 2:**
  Extend **App 1** by displaying a **box plot** from random generating normal distributed data in the **main panel**. Number of datapoints can be chosen apriori by the user in the **sidebar panel** (located at the left).

  Display:



Step 1: Maintain framework without text in sidebar and main panel

```
> install.packages("shiny")
> library(shiny)
> ui <- fluidPage(
        titlePanel(title="Second app ..."),
        sidebarLayout(
        sidebarPanel(),
        mainPanel())
        ))
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```

Step 2: Build the input structure in the sidebar panel

```
> install.packages("shiny")
> library(shiny)
> ui <- fluidPage(
        titlePanel(title="Second app ..."),
        sidebarLayout(
        sidebarPanel(numericInput(inputId
         ="n", "Sample size", value=10)),
        mainPanel())
        ))
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```

Step 3: Link the user inputs with R

```
> install.packages("shiny")
> library(shiny)
> ui <- fluidPage(
        titlePanel(title="Second app ..."),
        sidebarLayout(
        sidebarPanel(numericInput(inputId
         ="n", "Sample size", value=10)),
        mainPanel())
        ))
> server <- function(input , output){
        output$box<-renderPlot({
          boxplot(rnorm(input$n), col=
           "blue", main="Boxplot of normal
           generated data", xlab="Sample
           data")})}
> shinyApp(ui = ui , server = server)
```

Second app in the tutorial of Martial & Jeroen

Sample size

10

Step 4: Output the boxplot in the UI main panel

```
> install.packages("shiny")
> library(shiny)
> ui <- fluidPage(
        titlePanel(title="Second app ..."),
        sidebarLayout(
        sidebarPanel(numericInput(inputId
         ="n", "Sample size", value=10)),
        mainPanel(
          plotOutput(outputId="box")))
        ))
> server <- function(input , output){
        output$box<-renderPlot ({
          boxplot(rnorm(input$n), col =
          "blue", main="Boxplot of normal
          generated data", xlab="Sample
          data")})}
> shinyApp(ui = ui , server = server)
```

- **App 3:**
  Extend **App 2** by adding the possibility to choose your **own title** within a **text input** and **color** of the box plot with a **radio button** (in the **sidebar panel**). Additionally, a **submit button** needs to be present that only updates the main panel with a **click**.

  Display:

Step 1: Maintain framework of App 2

```
> install.packages("shiny")
> library(shiny)
> ui <- fluidPage(
        titlePanel(title="Third app ..."),
        sidebarLayout(
        sidebarPanel(numericInput(inputId
         ="n", "Sample size", value=10)),
        mainPanel(
          plotOutput(outputId="box")))
        ))
> server <- function(input , output){
        output$box<-renderPlot ({
          boxplot(rnorm(input$n), col =
          "blue", main="Boxplot of normal
          generated data", xlab="Sample
          data")})}
> shinyApp(ui = ui , server = server)
```

Step 2: Extend slider panel layout

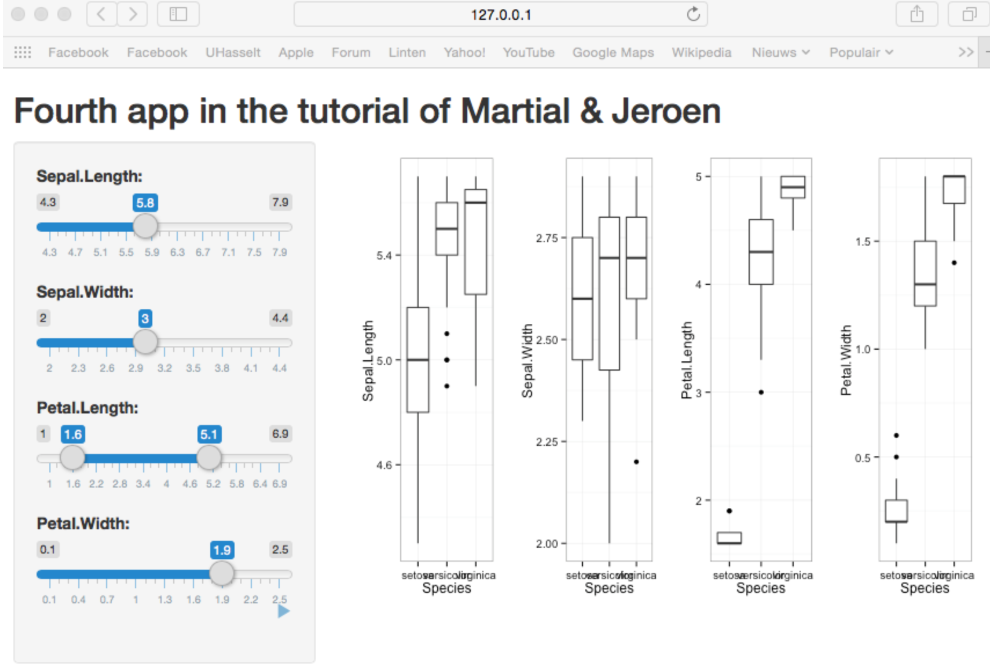
```
> ui <- fluidPage(
        titlePanel(title="Third app ..."),
        sidebarLayout(
        sidebarPanel(numericInput(inputId
         ="n", "Sample size", value=10),
         textInput(inputId="title", "Title
          of ...","Fancy Boxplot ..."),
         radioButtons(inputId="color",
          "Choose...",list("Blue","Green"),
          "Green"),
         submitButton("Apply changes")),
        mainPanel(
          plotOutput(outputId="box")))))
> server <- function(input , output){
        output$box<-renderPlot ({
          boxplot(rnorm(input$n), col =
          "blue", main="Boxplot of normal
          generated data", xlab="Sample
          data")})}
> shinyApp(ui = ui , server = server)
```

Step 3: Link the extra user inputs with R

```
> ui <- fluidPage(
        titlePanel(title="Third app ..."),
        sidebarLayout(
        sidebarPanel(numericInput(inputId
         ="n", "Sample size", value=10),
          textInput(inputId="title", "Title
          of ...","Fancy Boxplot ..."),
          radioButtons(inputId="color",
          "Choose ...",list("Blue","Green"),
          "Green"),
          submitButton("Apply changes")),
        mainPanel(
          plotOutput(outputId="box")))))
> server <- function(input , output){
        output$box<-renderPlot ({
          boxplot(rnorm(input$n), col =
          input$color,main=input$title,
          xlab="Sample data")})}
> shinyApp(ui = ui , server = server)
```

- **App 4:**
  Analyse the **irish dataset** by making a box plot of every numeric variable (i.e., Sepal.Length, Sepal.Width, Petal.Length, Petal.Width) per specie. Summarize these box plots into one figure. Different **slider inputs** need to be used for choosing your own subset of data.

  Display:



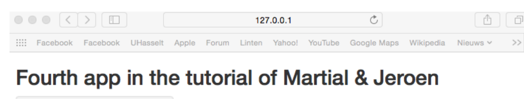
Step 1: Making libraries available & descriptive statistics **irish dataset**

```
> install.packages("shiny")
> library(shiny)
> library(ggplot2)
> library(gridExtra)
> str(iris)
> summary(iris)
```

```
> str(iris)
'data.frame':	150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species  
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50  
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50  
 Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50  
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199                  
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800                  
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500                  
```

Step 2: Maintain framework of **App 2**

```
> ui <- fluidPage(
         titlePanel(title="Fourth app ..."),
         sidebarLayout(
         sidebarPanel(),
         mainPanel())
         ))
> server <- function(input, output){}
> shinyApp(ui = ui, server = server)
```



Step 3: Import different slider inputs in UI slider panel

```
> ui <- fluidPage(
        titlePanel(title="Fourth app ..."),
        sidebarLayout(
        sidebarPanel(
         sliderInput(inputId="sepallength",
          "Sepal.Length:",min=4.3, max=7.9,
          value=5.8, step=0.1),
         sliderInput(inputId="sepalwidth",
          "Sepal.Width:", min=2, max=4.4,
          value=3, step=0.1),
         sliderInput(inputId="petallength",
          "Petal.Length:", min=1, max=6.9,
          value = c(1.6,5.1)),
         sliderInput(inputId="petalwidth",
          "Petal.Width:", min=0.1, max=2.5,
          value=1.3, step=0.3,
          animate = animationOptions(
            interval = 2600, loop = TRUE)),
        mainPanel())
        ))
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```

Step 4: Link the different slider inputs with R & obtain own subsets

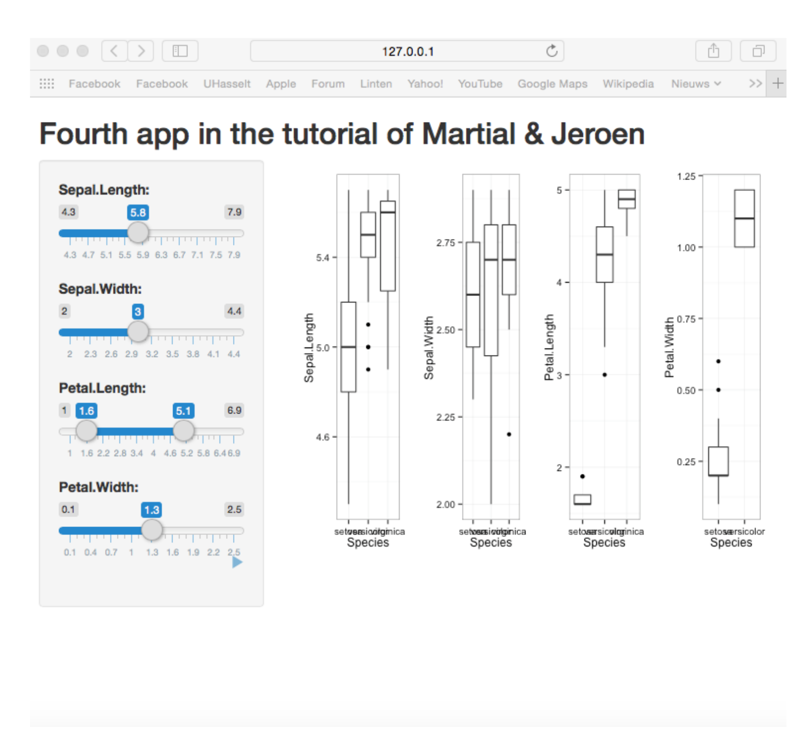
```
> ui <- fluidPage(
         titlePanel(title="Fourth app ..."),
         sidebarLayout(
         sidebarPanel(...),
         mainPanel()))
> server <- function(input , output){
output$box <- renderPlot({
    sepall<-subset(iris,Sepal.Length >= 4.3
         & Sepal.Length < input$sepallength,
         select=c(Sepal.Length, Species))
    sepalw<-subset(iris, Sepal.Width >= 2
         & Sepal.Width < input$sepalwidth,
         select=c(Sepal.Width, Species))
    petall<-subset(iris, Petal.Length>=
         input$petallength[1] & Petal.Length
         < input$petallength[2],
         select=c(Petal.Length, Species))
    petalw<-subset(iris, Petal.Width >= 0.1
         & Petal.Width < input$petalwidth,
         select=c(Petal.Width, Species))
  })}
> shinyApp(ui = ui , server = server)
```

Step 5: Create box plots in R & combine them into one graph

```
> ui <- fluidPage(
        titlePanel(title="Fourth app ..."),
        sidebarLayout(
        sidebarPanel(...),
        mainPanel()))
> server <- function(input, output){
  output$box<-renderPlot({
   sepall <- subset(...)
   sepalw <- subset(...)
   petall <- subset(...)
   petalw <- subset(...)
   test <- ggplot(sepall, aes(x = Species,
     y = Sepal.Length)) + geom_boxplot()
     + theme_bw()
   testb <- ggplot(sepalw, aes(x = Species,
     y = Sepal.Width)) + geom_boxplot() + theme_bw()
   testc <- ggplot(petall, aes(x = Species,
     y = Petal.Length)) + geom_boxplot()
     + theme_bw()
   testd <- ggplot(petalw, aes(x = Species,
     y = Petal.Width)) + geom_boxplot() + theme_bw()
   grid.arrange(test,testb,testc,testd,nrow=1)
   })}
> shinyApp(ui = ui, server = server)
```

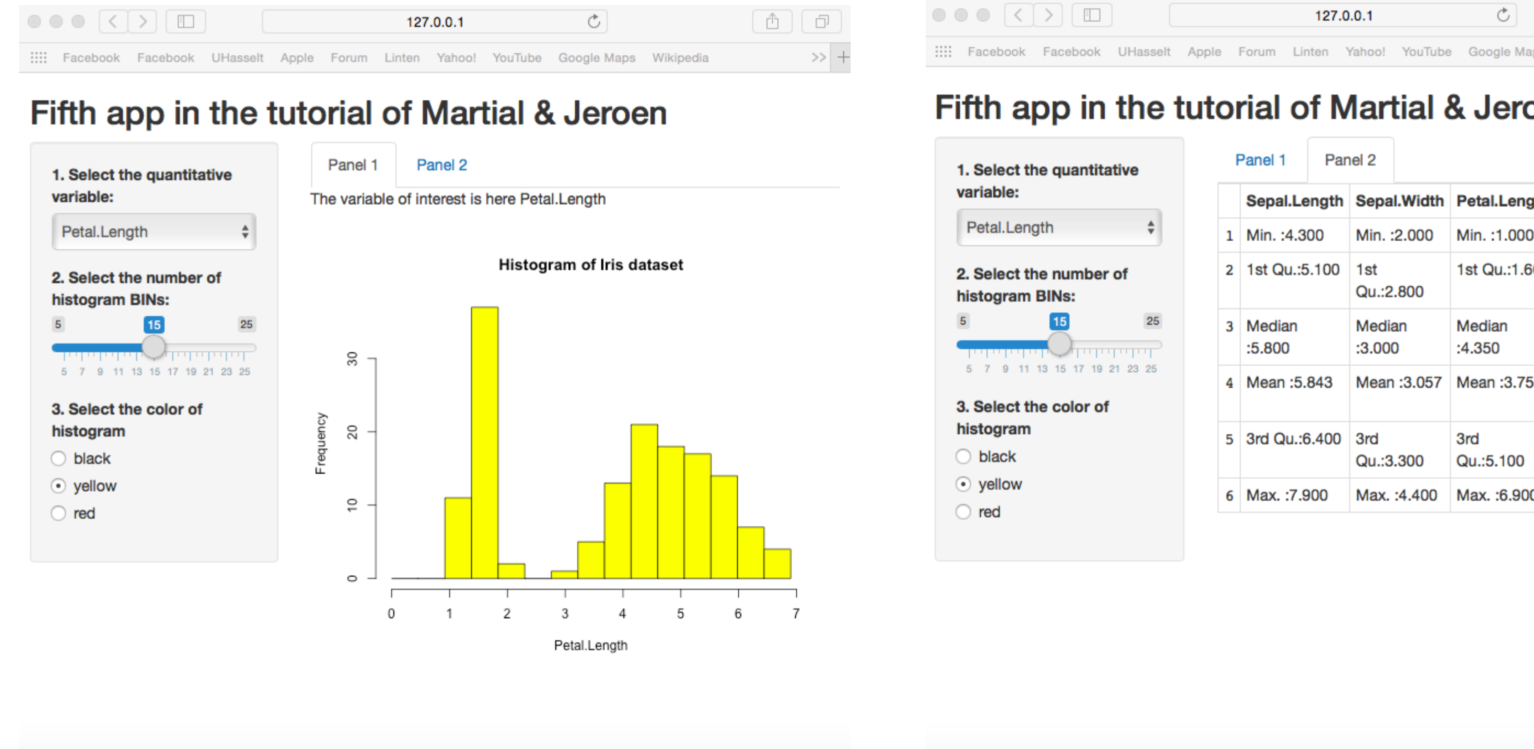
Step 6: Output the box plot in the UI main panel

```
> ui <- fluidPage(
         titlePanel(title="Fourth app ..."),
         sidebarLayout(
         sidebarPanel(...),
         mainPanel(
            plotOutput(outputId = "box")))
         )
> server <- function(input , output){
 output$box<-renderPlot ({
    sepall <- subset(...)
    sepalw <- subset(...)
    petall <- subset(...)
    petalw <- subset(...)
    test <- ggplot(...) + geom_boxplot()
      + theme_bw()
    testb <- ggplot(...) + geom_boxplot()
      + theme_bw()
    testc <- ggplot(...) + geom_boxplot()
      + theme_bw()
    testd <- ggplot(...) + geom_boxplot()
      + theme_bw()
    grid.arrange(...)})}
> shinyApp(ui = ui , server = server)
```

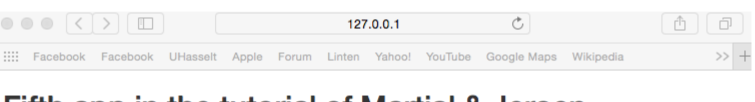


- **App 5:**
  Analyse the **irish dataset** by creating **two tabsets** in the **main panel**. The first one contains a histogram per chosen variable, while the second one displays a summary output of all variables. Variable selection is obtained by a **select input**.

  Display:



Step 1: Create basic layout structure




```
> ui <- fluidPage(
        titlePanel(title="Fifth app ..."),
        sidebarLayout(
        sidebarPanel(),
        mainPanel())
        )
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```

Step 2: Build the input structure in the sidebar panel

```
> ui <- fluidPage(
         titlePanel(title="Fifth app ..."),
         sidebarLayout(
         sidebarPanel(
          selectInput(inputId="var","1.
             Select ...", choices=c(
             "Sepal.Length"=1, "Sepal.Width"=
             2,"Petal.Length"=3,"Petal.Width"
             =4),selected=3,selectize=FALSE),
          sliderInput(inputId="bin","2. Sele
             ...", min=5, max=25, value=15),
          radioButtons(inputId="colour",
             label="3. Select ...",choices=c(
             "black","yellow","red"),selected
             ="yellow")
         ),
         mainPanel())
         )
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```

Fifth app in the tutorial of Martial & Jeroen

1. Select the quantitative variable:

Petal.Length

2. Select the number of histogram BINs:

5 15 25

3. Select the color of histogram

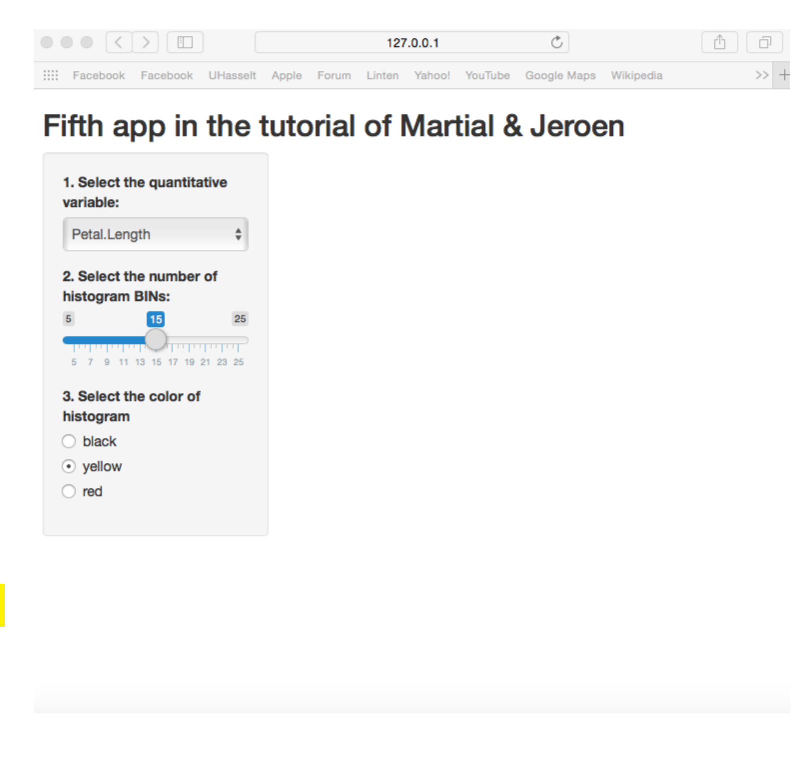
black

yellow

red

Step 3: Link the user inputs with R

```
> ui <- fluidPage(
         titlePanel(title="Fifth app ..."),
         sidebarLayout(
         sidebarPanel(
          selectInput(...),
          sliderInput(...),
          radioButtons(...)),
         mainPanel()))
> server <- function(input , output){
        output$text1 <- renderText({
           colm = as.numeric(input$var)
           paste("The ...", names(iris[colm]))})
        output$myhist <- renderPlot({
           colm = as.numeric(input$var)
           hist(iris[,colm], col =input$colour,
              xlim = c(0, max(iris[,colm])),
              main = "Histogram ...", breaks =
              seq(0, max(iris[,colm]),l=input$bin+1),
              xlab = names(iris[colm]))})
        output$summary <- renderTable({
             summary(iris)})
 }
> shinyApp(ui = ui , server = server)
```



Step 4: Create tabsets with outputs in the UI main panel

```
> ui <- fluidPage(
         titlePanel(title="Fifth app ..."),
         sidebarLayout(
         sidebarPanel(
          selectInput(...),
          sliderInput(...),
          radioButtons(...)),
         mainPanel(
          tabsetPanel(type="tab",
            tabPanel("Panel 1",
              textOutput(outputId="text1"),
              plotOutput(outputId="myhist")),
            tabPanel("Panel 2",
              tableOutput(outputId="summary")
   )))
> server <- function(input, output){
       output$text1 <- renderText({...})
       output$myhist <- renderPlot({...})
       output$summary <- renderTable({...})
       }
> shinyApp(ui = ui, server = server)
```

Fifth app in the tutorial of Martial & Jeroen

1. Select the quantitative variable: Petal.Length

2. Select the number of histogram BINs: 15

3. Select the color of histogram: black / yellow / red

Panel 1 | Panel 2

The variable of interest is here Petal.Length

Histogram of Iris dataset (Frequency vs Petal.Length)

3.2 Focus on special reactive functions

- Until now, **render*() functions** such as **renderText()**, **renderPlot()**, **renderTable()** have been used (in the server statement)

- These functions can be placed under the **set of reactive functions**, i.e., functions that react when the user selection changes through the input widgets

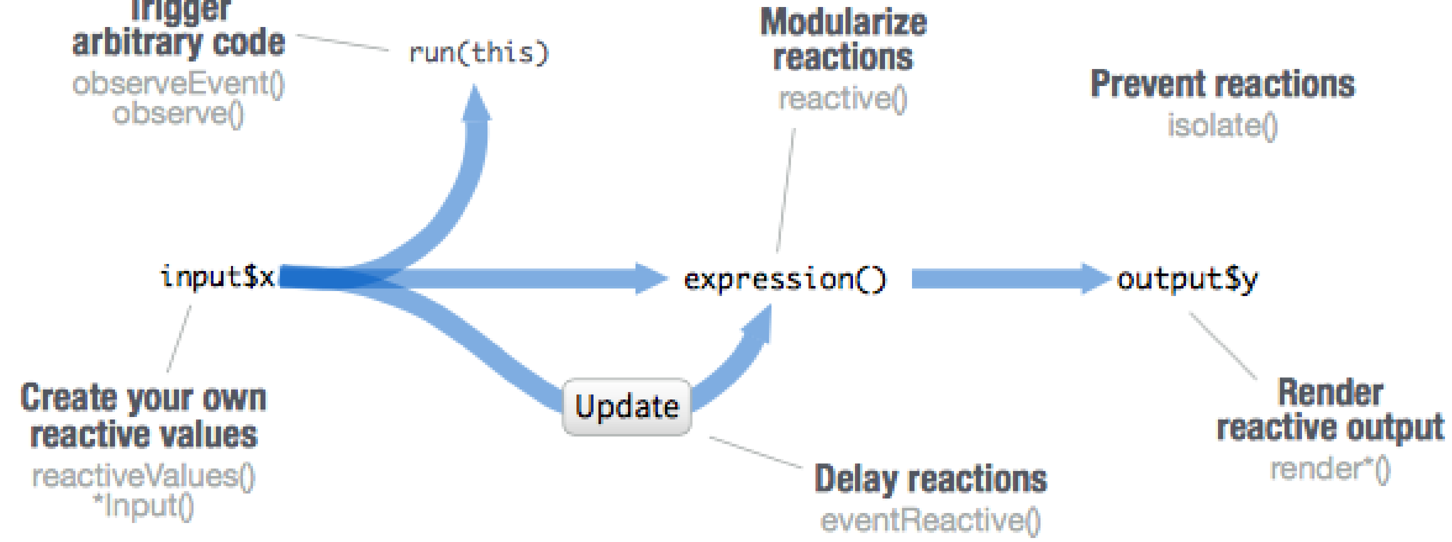
- Another function in shiny having the reactive property is the **reactive() function**.
  - **Functionality:** Any expression given in the reactive function that depends on the input variable would change (rather updates or re-evaluates) with any change in the input variable

  - **Usuability:** When using reactive expressions, i.e., when the expression is

dependent on input variable, and there is need for the expression to be reactive

- **Advantage:** Reusability, i.e., to evaluate an expression once and its value could be used within multiple render statements. That way the reactive expression need not to be calculated multiple time in each render statement.

- **Overview** of reactive functions:



Create your own reactive values

```
library(shiny)

ui <- fluidPage(
 textInput("a","")
)

server <-
function(input,output){
 rv <- reactiveValues()
 rv$number <- 5
}

shinyApp(ui, server)
```

***Input() functions**
(see front page)

reactiveValues(...)

Each input function creates a reactive value stored as **input$<inputId>**

reactiveValues() creates a list of reactive values whose values you can set.

Render reactive output

```
library(shiny)
ui <- fluidPage(
 textInput("a","")
)

server <-
function(input,output){
 output$b <-
  renderText({
   input$a
  })
}

shinyApp(ui, server)
```

render*() functions
(see front page)

Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.

Save the results to **output$<outputId>**

Prevent reactions

```
library(shiny)
ui <- fluidPage(
 textInput("a",""),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   isolate({input$a})
  })
}

shinyApp(ui, server)
```

isolate(expr)

Runs a code block. Returns a **non-reactive** copy of the results.

Trigger arbitrary code

```
library(shiny)
ui <- fluidPage(
 textInput("a",""),
 actionButton("go", "")
)

server <-
function(input,output){
 observeEvent(input$go,
   print(input$a)
  })
}

shinyApp(ui, server)
```

observeEvent(eventExpr, **handlerExpr**, event.env, event.quoted, handler.env, handler.quoted, labe, suspended, priority, domain, autoDestroy, ignoreNULL)

Runs code in 2nd argument when reactive values in 1st argument change. See **observe()** for alternative.

Modularize reactions

```
library(shiny)
ui <- fluidPage(
 textInput("a",""),
 textInput("z", "")
)

server <-
function(input,output){
 re <- reactive({
  paste(input$a,input$b})
 output$b <- renderText({
   re()
  })
}
shinyApp(ui, server)
```

reactive(x, env, quoted, label, domain**)**

Creates a **reactive expression** that

- caches its value to reduce computation
- can be called by other code
- notifies its dependencies when it ha been invalidated

Call the expression with function syntax, e.g. `re()`

Delay reactions

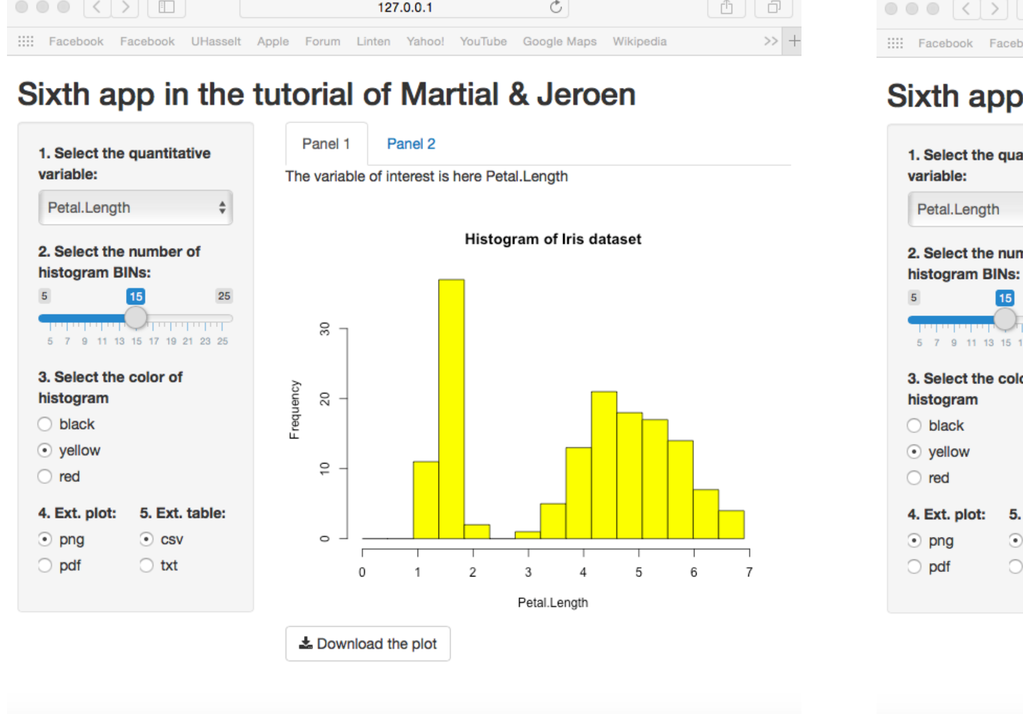
```
library(shiny)
ui <- fluidPage(
 textInput("a",""),
 actionButton("go", "")
)

server <-
function(input,output){
 re <- eventReactive(
  input$go,{input$a})
 output$b <- renderText({
   re()
  })
}

shinyApp(ui, server)
```

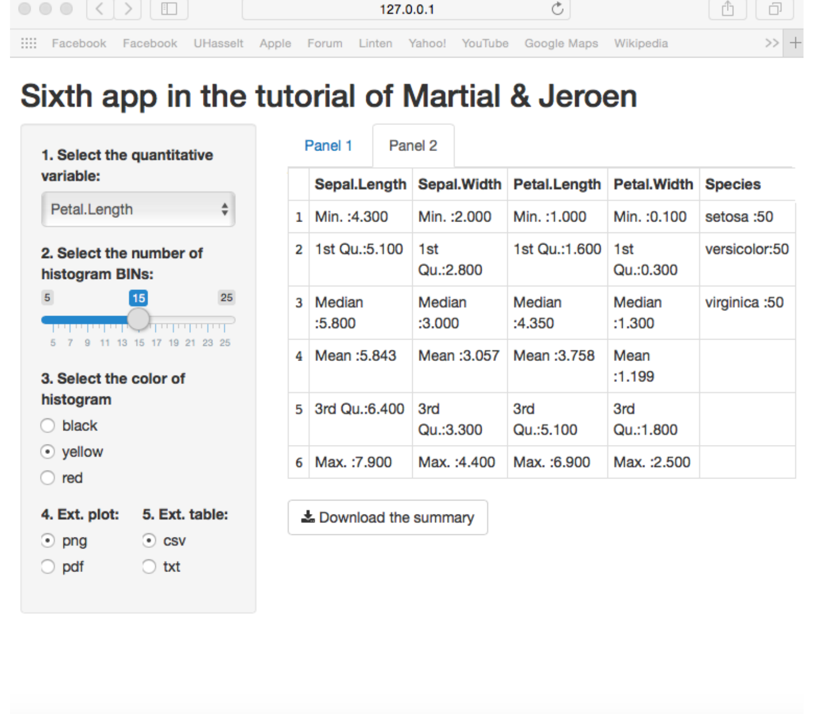
eventReactive(eventExpr, **valueExpr**, event.env, event.quoted, value.env, value.quoted, label, domain, ignoreNULL**)**

Creates reactive expression with code in 2nd argument that only invalidates when reactive values in 1st argument change.

- **App 6:**
  Use the **reactive function** (in the server statement of **App 5**) for the **quantitative variable selection**, and add two download buttons in the tabset panels that enables the user to download the histogram (Panel 1) and summary table (Panel 2), respectively. Different extensions can be chosen in the sider panel by radio buttons.

  Display:

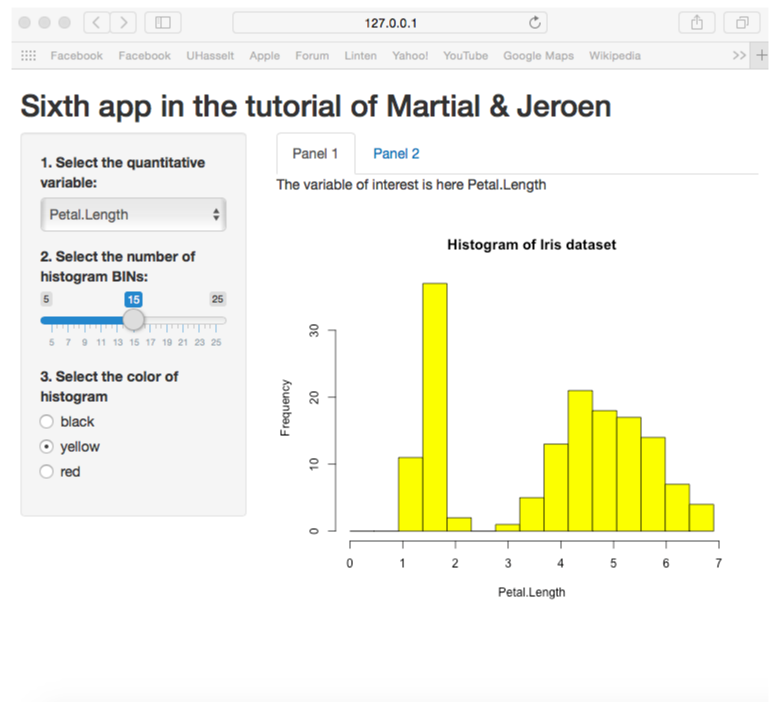




	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	Min. :4.300	Min. :2.000	Min. :1.000	Min. :0.100	setosa :50
2	1st Qu.:5.100	1st Qu.:2.800	1st Qu.:1.600	1st Qu.:0.300	versicolor:50
3	Median :5.800	Median :3.000	Median :4.350	Median :1.300	virginica :50
4	Mean :5.843	Mean :3.057	Mean :3.758	Mean :1.199	
5	3rd Qu.:6.400	3rd Qu.:3.300	3rd Qu.:5.100	3rd Qu.:1.800	
6	Max. :7.900	Max. :4.400	Max. :6.900	Max. :2.500	

Step 1: Maintain coding structure of **App 5**

```
> ui <- fluidPage(
         titlePanel(title="Sixth app ..."),
         sidebarLayout(
         sidebarPanel(
          selectInput(...),
          sliderInput(...),
          radioButtons(...)),
         mainPanel(...))
         )
> server <- function(input , output){
        output$text1 <- renderText({
          colm = as.numeric(input$var)
          paste("The ...", names(iris[colm]))
        })
        output$myhist <- renderPlot({
          colm = as.numeric(input$var)
          hist(iris[,colm], col =input$colour,
            xlim = c(0, max(iris[,colm])),
            main = "Histogram ...", breaks =
            seq(0, max(iris[,colm]),l=input$bin+1),
            xlab = names(iris[colm]))
        })
        output$summary <- renderTable({
           summary(iris)
        })
}
> shinyApp(ui = ui , server = server)
```



Step 2: Change the defined colm expression by a reactive function

```
> ui <- fluidPage(
         titlePanel(title="Sixth app ..."),
         sidebarLayout(
         sidebarPanel(
          selectInput(...),
          sliderInput(...),
          radioButtons(...)),
         mainPanel(...))
         )
> server <- function(input , output){
        colm <- reactive({
          as.numeric(input$var)
        })
        output$text1 <- renderText({
          paste("The ...", names(iris[colm()]))
        })
        output$myhist <- renderPlot({
          hist(iris[,colm()], col =input$colour,
            xlim = c(0, max(iris[,colm()])),
            main = "Histogram ...", breaks =
            seq(0, max(iris[,colm()]),l=input$bin+1),
            xlab = names(iris[colm()]))
        })
        output$summary <- renderTable({
           summary(iris)
        })
  }
> shinyApp(ui = ui , server = server)
```

Step 3: Add the file extensions to the sider panel

```
> ui <- fluidPage(
        titlePanel(title="Sixth app ..."),
        sidebarLayout(
        sidebarPanel(
         selectInput(...),
         sliderInput(...),
         radioButtons(...),
         splitLayout(
          radioButtons(inputId="plotext",
           label="4. Ext. plot:",choices=c("png",
           "pdf"), selected="png"),
          radioButtons(inputId="tableext",
           label="5. Ext. table:",choices=c("csv",
           "txt"), selected="csv"))
         ),
        mainPanel(...))
        )
> server <- function(input , output){
       colm <- reactive({...})
       output$text1 <- renderText({...})
       output$myhist <- renderPlot({...})
       output$summary <- renderTable({...})
}
> shinyApp(ui = ui , server = server)
```

127.0.0.1

Facebook Facebook UHasselt Apple Forum Linten Yahoo! YouTube Google Maps Wikipedia

Sixth app in the tutorial of Martial & Jeroen

1. Select the quantitative variable:

Petal.Length

2. Select the number of histogram BINs:

5 15 25

3. Select the color of histogram

black

yellow

red

4. Ext. plot: 5. Ext. table:

png csv

pdf txt

Panel 1 Panel 2

The variable of interest is here Petal.Length

Histogram of Iris dataset

Frequency

Petal.Length

Step 4: Link the plot file extension with R

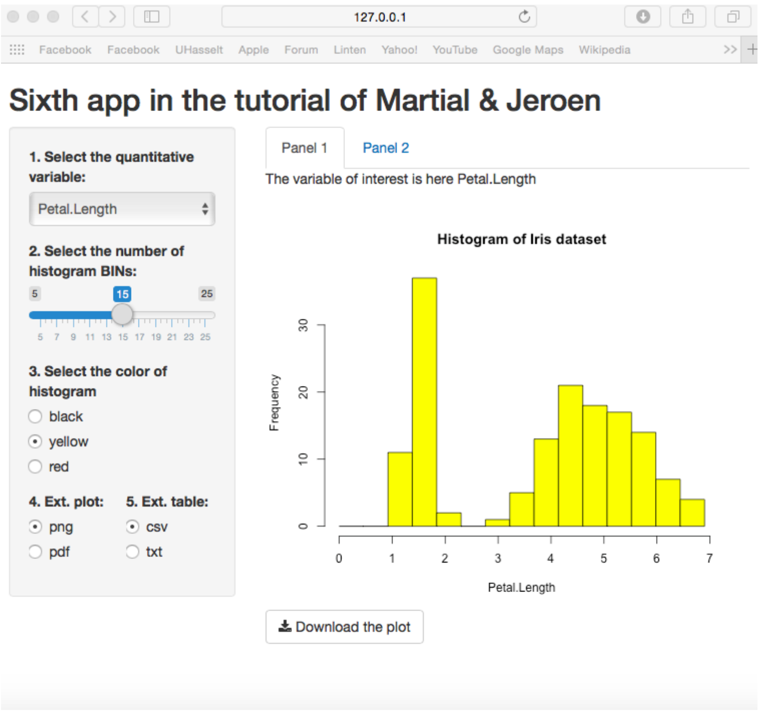
```
> ui <- fluidPage(
         titlePanel(title="Sixth app ..."),
         sidebarLayout(
         sidebarPanel(
          selectInput(...),
          sliderInput(...),
          radioButtons(...),
          splitLayout(...)),
         mainPanel(...)))
> server <- function(input , output){
       colm <- reactive({...})
       output$text1 <- renderText({...})
       output$myhist <- renderPlot({...})
       output$summary <- renderTable({...})
       output$downplot <- downloadHandler(
         filename=function(){
          paste("iris_hist",input$plotext,sep=".")},
         content=function(file){
           if(input$plotext=="png")
              png(file)
           else
              pdf(file)
           hist(iris[,colm()], ...)
           dev.off()}
         )
}
> shinyApp(ui = ui , server = server)
```

Step 5: Link the table file extension with R

```
> ui <- fluidPage(
          titlePanel(title="Sixth app ..."),
          sidebarLayout(
          sidebarPanel(
           selectInput(...),
           sliderInput(...),
           radioButtons(...),
           splitLayout(...)),
          mainPanel(...))
          )
> server <- function(input , output){
        colm <- reactive({...})
        output$text1 <- renderText({...})
        output$myhist <- renderPlot({...})
        output$summary <- renderTable({...})
        output$downplot <- downloadHandler(...)
        output$downsum <- downloadHandler(
          filename=function(){
           paste("iris_sum",input$tableext,sep=".")},
          content=function(file){
           sep <- switch(input$tableext, "csv"=",",
              "txt"=";")
           write.table(summary(iris), file, sep=sep)}
        )
}
> shinyApp(ui = ui , server = server)
```

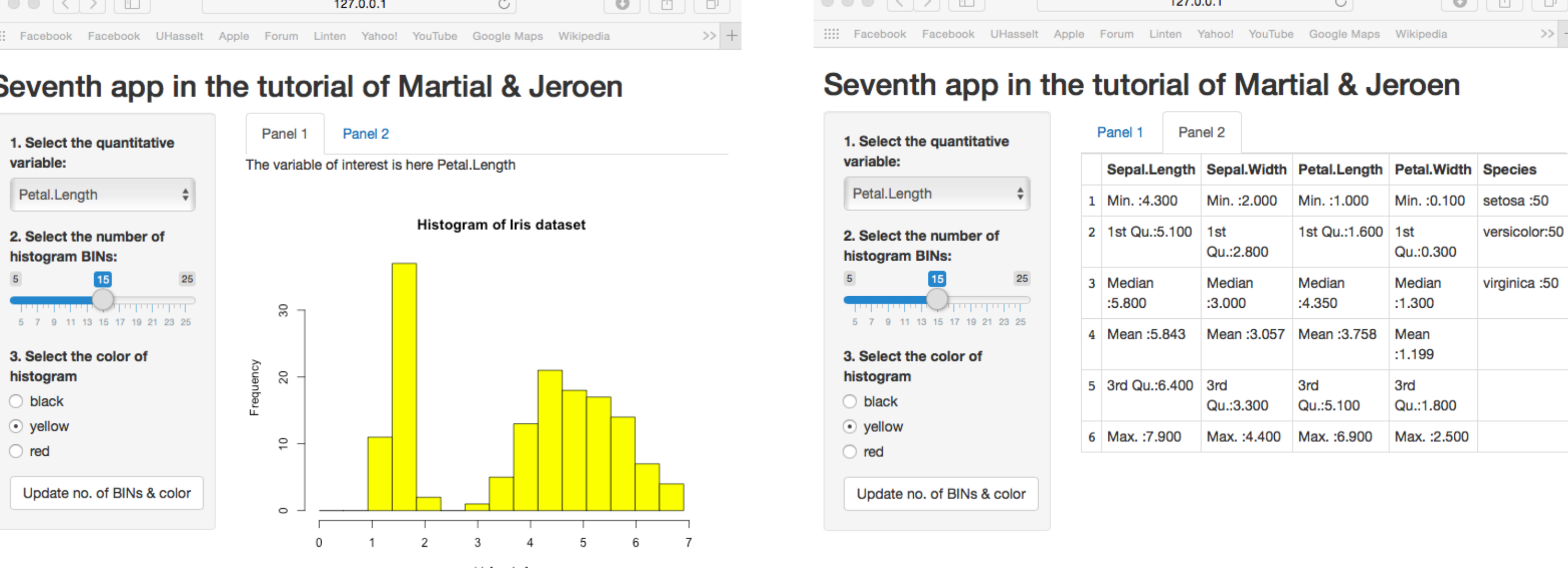
Step 6: Create download buttons with outputs in the UI tabset panels

```
> ui <- fluidPage(
        titlePanel(title="Sixth app ..."),
        sidebarLayout(
        sidebarPanel(
         selectInput(...),
         sliderInput(...),
         radioButtons(...),
         splitLayout(...)),
        mainPanel(
         tabsetPanel(type="tab",
          tabPanel("Panel 1",
           textOutput(outputId="text1"), br(),
           plotOutput(outputId = "myhist"),
           downloadButton(outputId="downplot",
              label="Download the plot")),
             tabPanel("Panel 2",
             tableOutput(outputId="summary"),
             downloadButton(outputId="downsum",
               label="Download the summary"))))))
> server <- function(input , output){
      colm <- reactive({...})
      output$text1 <- renderText({...})
      output$myhist <- renderPlot({...})
      output$summary <- renderTable({...})
      output$downplot <- downloadHandler(...)
      output$downsum <- downloadHandler(...)
}
> shinyApp(ui = ui , server = server)
```



- **App 7:**
  Use **partial reactivity** in **App 5** to change the number BIN's and color of the histogram. An **action button** in the sider panel needs to be present to occur this reactivity.

  Display:



Step 1: Maintain coding structure of App 5

```
> ui <- fluidPage(
          titlePanel(title="Seventh app ...")
          sidebarLayout(
          sidebarPanel(
           selectInput(...),
           sliderInput(...),
           radioButtons(...)),
          mainPanel(
           tabsetPanel(type="tab",
             tabPanel("Panel 1", ...),
             tabPanel("Panel 2", ...))
    )))
> server <- function(input , output){
        output$text1 <- renderText({...})
        output$myhist <- renderPlot({...})
        output$summary <- renderTable({...})
        }
> shinyApp(ui = ui , server = server)
```

Step 2: Add the **action button** to the sider panel

```
> ui <- fluidPage(
        titlePanel(title="Seventh app ..."),
        sidebarLayout(
        sidebarPanel(
         selectInput(...),
         sliderInput(...),
         radioButtons(...),
         actionButton(inputId="action",
          label="Update ...")),
        mainPanel(
         tabsetPanel(type="tab",
           tabPanel("Panel 1", ...),
           tabPanel("Panel 2", ...))
   )))
> server <- function(input, output){
       output$text1 <- renderText({...})
       output$myhist <- renderPlot({...})
       output$summary <- renderTable({...})
       }
> shinyApp(ui = ui, server = server)
```

Step 3: Link the action button with R to obtain the required partial reactivity

```
> ui <- fluidPage(
        titlePanel(title="Seventh app ..."),
        sidebarLayout(
        sidebarPanel(
         selectInput(...),
         sliderInput(...),
         radioButtons(...),
         actionButton(inputId="action",
          label="Update ...")),
        mainPanel(
         tabsetPanel(type="tab",
           tabPanel("Panel 1", ...),
           tabPanel("Panel 2", ...))))))
> server <- function(input, output){
       colm <- reactive({...})
       output$text1 <- renderText({...})
       output$myhist <- renderPlot({
         input$action
         colm = as.numeric(input$var)
         hist(iris[,colm],col=isolate(input$colour),
          xlim=c(0, max(iris[,colm])), main=
          "Histogram ...", breaks=seq(0,
          max(iris[,colm]),l=isolate(input$bin+1)),
          xlab = names(iris[colm]))})
       output$summary <- renderTable({...})
  }
> shinyApp(ui = ui, server = server)
```

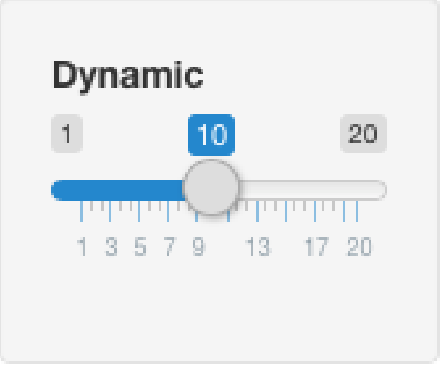
3.3 Progress dynamic user interface

- User interface panels like the sider panel are often kept static in the app

- **Dynamic user interfaces** can be obtained as well
  - **Basic idea:** Input statements depend on other input statements, and will change when the user chooses a different option.

  - **Implementation:** The user interface components are generated as HTML on the server inside a **renderUI()** block and sent to the client, which displays them with **uiOutput()**. Each time a new component is sent to the client, it completely replaces the previous component.

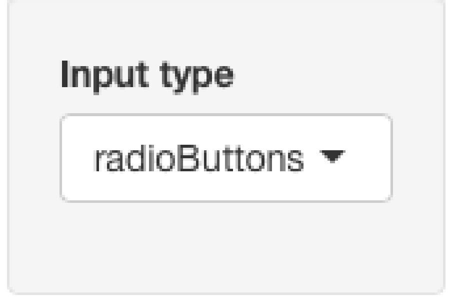
- **Example:**



slider

Dynamic input value:

int 10



Option 1

Option 2

Input type:

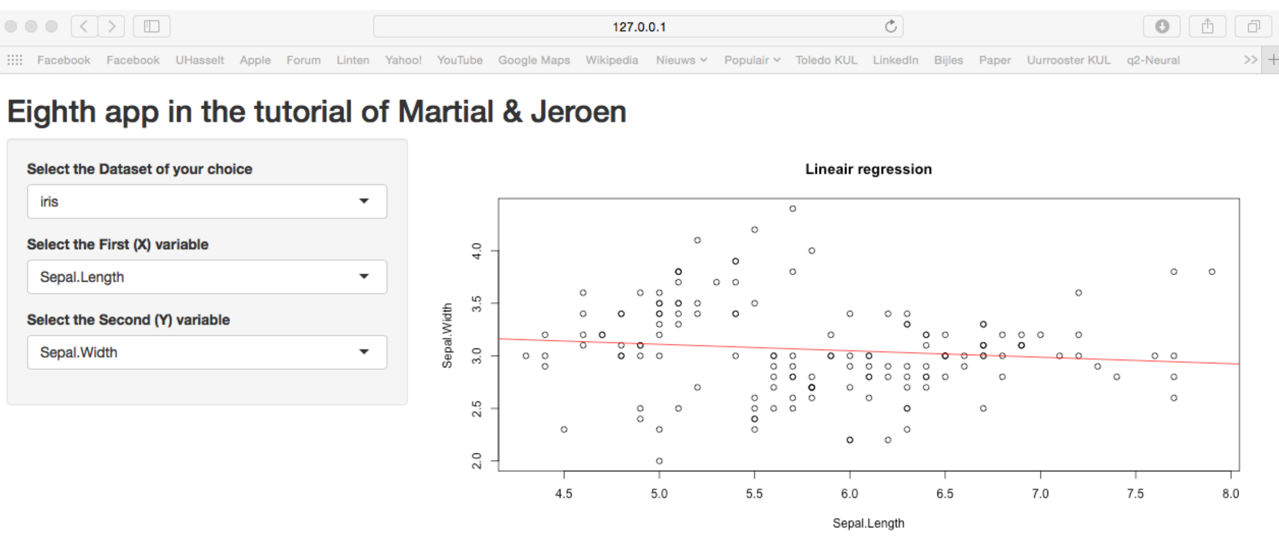
radioButtons

Dynamic input value:

chr "option2"

- **App 8:**
  Obtain dynamic user interface components in such a way that a scatter plot with lineair regression (Main panel) can be made on different variables (Slider panel) of a particular dataset. These variables depend on their corresponding dataset, and will be present in the slider panel when the right dataset is chosen (Slider panel).

  Display:



Step 1: Global coding structure & add selection input of three datasets

```
> ui <- fluidPage(
         titlePanel(title="Eighth app ..."),
         sidebarLayout(
         sidebarPanel(
          selectInput(inputId="data1",
           label="Select ...", choices=
           c("iris","mtcars","trees"))),
         mainPanel(
    )))
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```

127.0.0.1
Facebook Facebook UHasselt Apple Forum Linten Yahoo! YouTube Google Maps Wikipedia

Eighth app in the tutorial of Martial & Jeroen

Select the Dataset of your choice

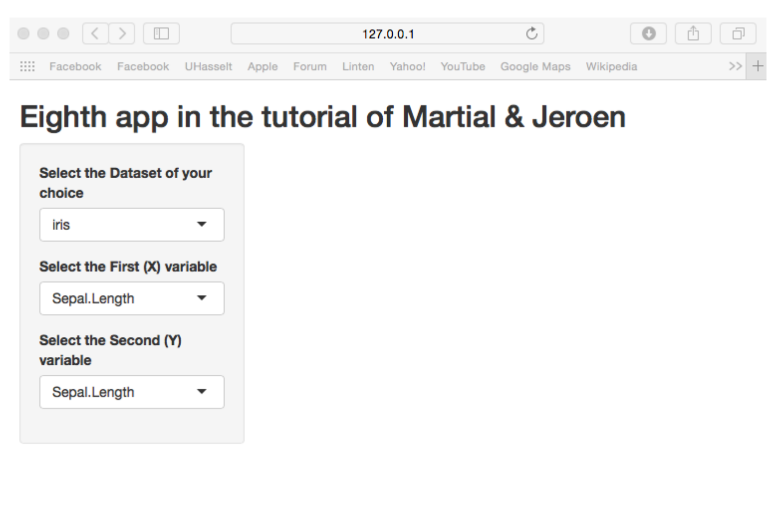
iris

Step 2: Create reactive function with variable names & use **renderUI()**

```
> ui <- fluidPage(
        titlePanel(title="Eighth app ..."),
        sidebarLayout(
        sidebarPanel(
         selectInput(inputId="data1", ...))
        mainPanel(
   )))
> server <- function(input , output){
       var <- reactive({
          switch(input$data1,
             "iris"  = names(iris),
             "mtcars"  = names(mtcars),
             "trees"  = names(trees))
          })
       output$vx <- renderUI({
         selectInput("variablex", "Select
           ...", choices = var())
         })
       output$vy <- renderUI({
         selectInput("variabley", "Select
           ...", choices = var())
         })
}
> shinyApp(ui = ui , server = server)
```

Step 3: Link renderUI() to UI with uiOutput()

```
> ui <- fluidPage(
         titlePanel(title="Eighth app ..."),
         sidebarLayout(
         sidebarPanel(
          selectInput(inputId="data1", ...),
          uiOutput("vx"),
          uiOutput("vy")),
         mainPanel(
     )))
> server <- function(input , output){
       var <- reactive({
          switch(input$data1,
             "iris" = names(iris),
             "mtcars" = names(mtcars),
             "trees" = names(trees))
          })
       output$vx <- renderUI({
         selectInput("variablex", "Select
           ...", choices = var())
          })
       output$vy <- renderUI({
         selectInput("variabley", "Select
           ...", choices = var())
          })
}
> shinyApp(ui = ui , server = server)
```



Step 4: Link the three selection inputs with plot in R

```
> ui <- fluidPage(
        titlePanel(title="Eighth app ..."),
        sidebarLayout(
        sidebarPanel(
         selectInput(inputId="data1", ...),
         uiOutput("vx"),
         uiOutput("vy")),
        mainPanel(
    )))
> server <- function(input , output){
      var <- reactive({...})
      output$vx <- renderUI({...})
      output$vy <- renderUI({...})
      output$plot <- renderPlot({
        attach(get(input$data1))
        lm.out = lm(get(input$variabley) ~
         get(input$variablex),data=get(input$data1))
        plot(get(input$variabley) ~
         get(input$variablex),data=get(input$data1),
         xlab=input$variablex, ylab=input$variabley,
         main = "Lineair regression")
        abline(lm.out, col="red")
})
}
> shinyApp(ui = ui , server = server)
```

127.0.0.1

Facebook Facebook UHasselt Apple Forum Linten Yahoo! YouTube Google Maps Wikipedia

Eighth app in the tutorial of Martial & Jeroen

Select the Dataset of your choice

iris

Select the First (X) variable

Sepal.Length

Select the Second (Y) variable

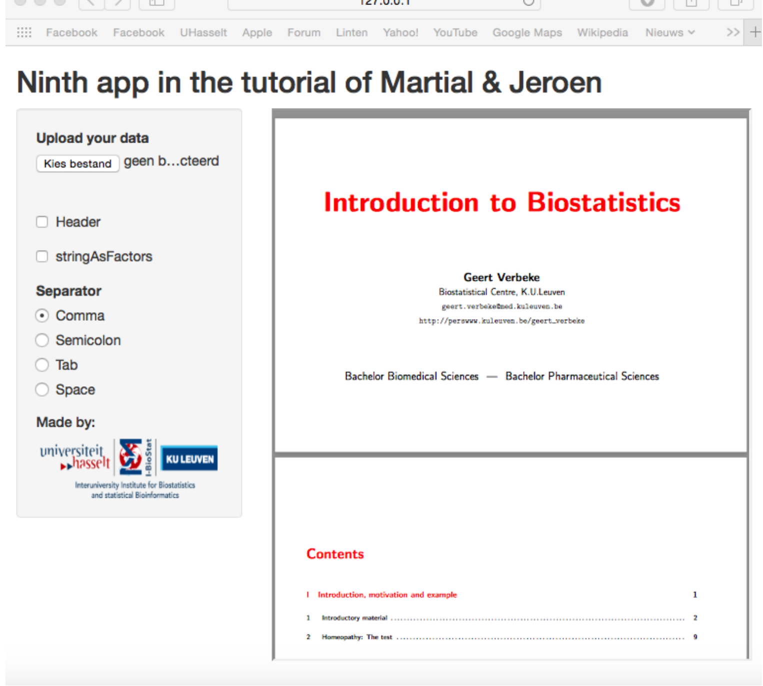
Sepal.Length

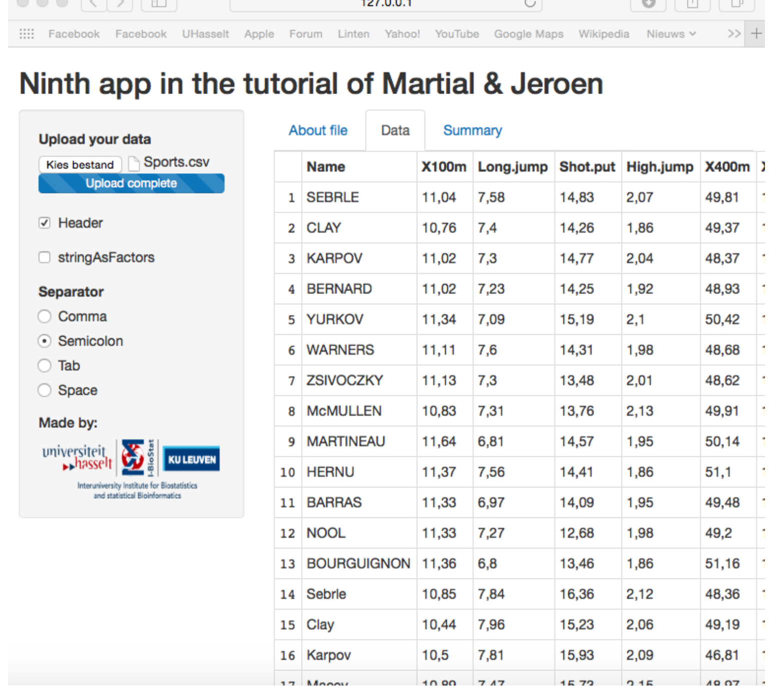
Step 5: Output plot in UI

```
> ui <- fluidPage(
        titlePanel(title="Eighth app ..."),
        sidebarLayout(
        sidebarPanel(
         selectInput(inputId="data1", ...),
         uiOutput("vx"),
         uiOutput("vy")),
        mainPanel(plotOutput("plot")
   )))
> server <- function(input, output){
      var <- reactive({...})
      output$vx <- renderUI({...})
      output$vy <- renderUI({...})
      output$plot <- renderPlot({...})
}
> shinyApp(ui = ui, server = server)
```

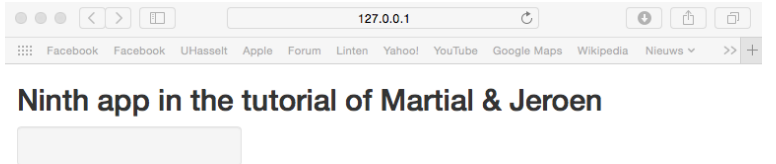
- **App 9:**
  Create an app where the user can upload their own data file in the slider panel and outputs information (About; Data; Summary) of this file in the main panel. When no data is selected by the user, a pdf format is displayed in the main panel. In addition, an image/logo needs to be present in the slider panel, together with the ability to select different settings that specifies the data structure.

  Display:





Step 1: Create basic layout structure




```
> install.packages("shiny")
> library(shiny)
> library(ggplot2)
> library(lattice)
> ui <- fluidPage(
        titlePanel(title="Ninth app ..."),
        sidebarLayout(
        sidebarPanel(),
        mainPanel()))
> server <- function(input, output){}
> shinyApp(ui = ui, server = server)
```

Step 2: Build the input structure in the sidebar panel

```
> ui <- fluidPage(titlePanel(title="..."),
        sidebarLayout(
        sidebarPanel(
         fileInput(inputId="file", label=
          "Upload your data"),
         checkboxInput(inputId="header",
          label="Header", value=FALSE),
         checkboxInput(inputId=
          "stringAsFactors", label=
          "stringAsFactors", value=FALSE),
         radioButtons(inputId="sep", label=
          "Separator",choices=c(Comma=',',
          Semicolon=';',Tab='\t',Space=''),
          selected = ','),
         h5("Made by:"),
         div(tags$img(style="height:60px;
          width:180px", src="http://www.
          uhasselt.be/images/logos/
          instituten/IBiostat-logo.png"),
          style="text-align: center;")),
        mainPanel()))
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```

Ninth app in the tutorial of Martial & Jeroen

Upload your data

Kies bestand geen b...cteerd

Header

stringAsFactors

Separator

Comma

Semicolon

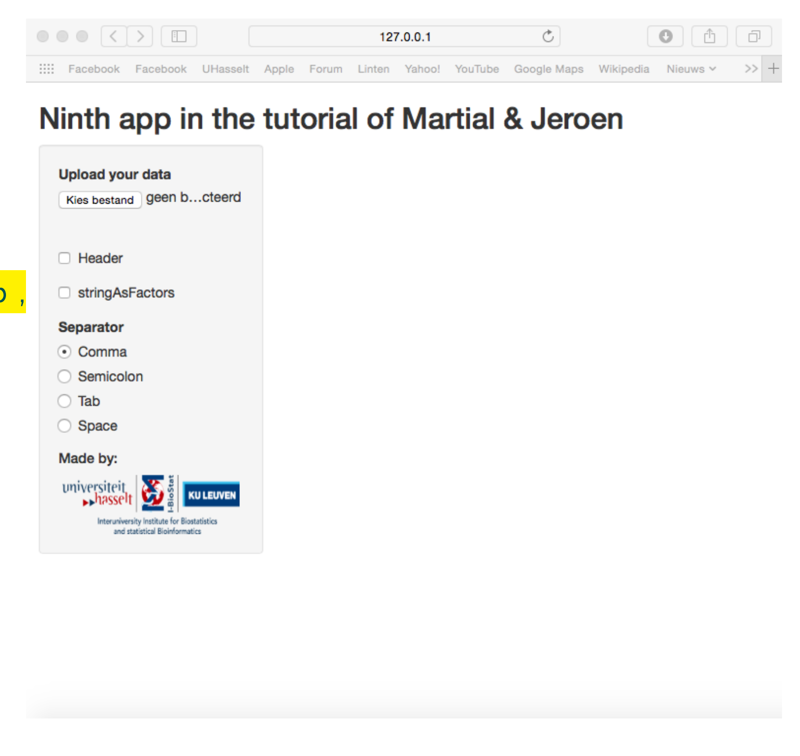
Tab

Space

Made by:

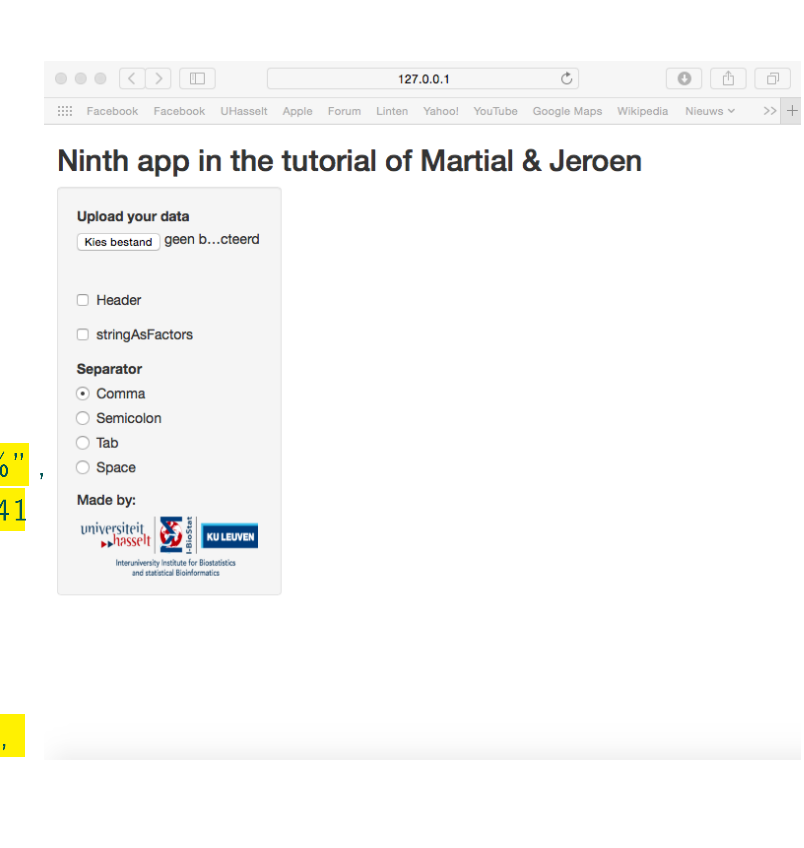
Step 3: Link the user inputs with R

```
> ui <- fluidPage(titlePanel(title="..."),
         sidebarLayout(
         sidebarPanel(
          fileInput(...), checkboxInput(...),
          checkboxInput(...), radioButtons(...),
          h5(...), div(tags$img(...)),
         mainPanel()))
> server <- function(input , output){
      data <- reactive({
       file1 <- input$file
       if(is.null(file1)){return()}
       read.table(file=file1$datapath,sep=input$sep,
         header=input$header,stringsAsFactors=
         input$stringsAsFactors)})
      output$filedf <- renderTable({
       if(is.null(data())){return()}
       input$file})
      output$sum <- renderTable({
       if(is.null(data())){return()}
       summary(data())})
      output$table <- renderTable({
       if(is.null(data())){return()}
       data()})
}
> shinyApp(ui = ui , server = server)
```



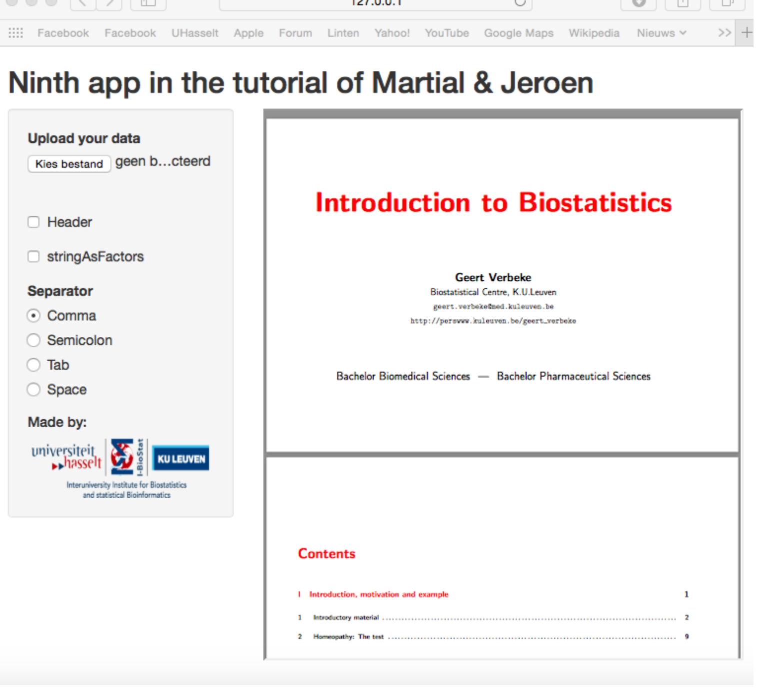
Step 4: Use renderUI()

```
> ui <- fluidPage(titlePanel(title="..."),
        sidebarLayout(
        sidebarPanel(
         fileInput(...),
         checkboxInput(...),checkboxInput(...),
         radioButtons(...),
         h5(...),div(tags$img(...)),
        mainPanel()))
> server <- function(input , output){
      data <- reactive({...})
      output$filedf <- renderTable({...})
      output$sum <- renderTable({...})
      output$table <- renderTable({...})
      output$tb <- renderUI({
       if(is.null(data())){
        tags$iframe(style="height:560px;width:100%",
         src="https://perswww.kuleuven.be/~u0018341
         /documents/biomedwet_farmacie.pdf")}
       else
        tabsetPanel(tabPanel("About file",
         tableOutput("filedf")),tabPanel("Data",
         tableOutput("table")),tabPanel("Summary",
         tableOutput("sum")))
       })
}
> shinyApp(ui = ui , server = server)
```



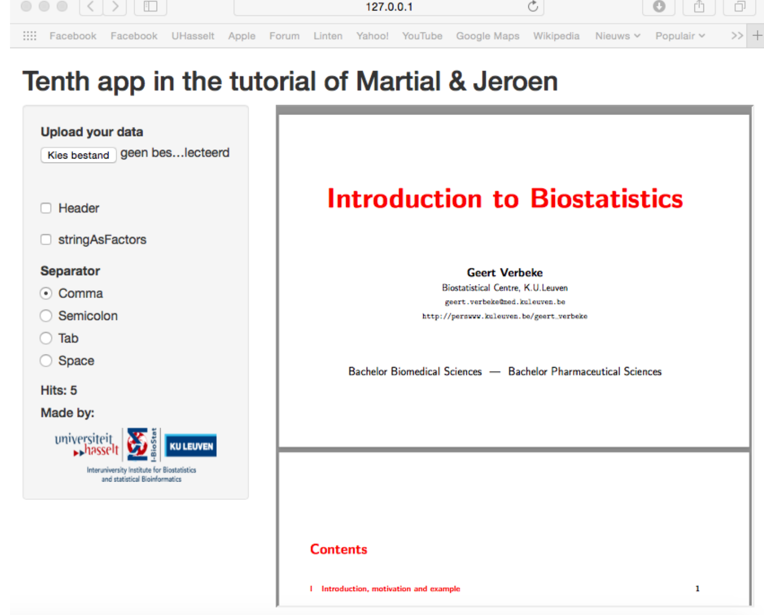
Step 5: Link renderUI() to UI with uiOutput()

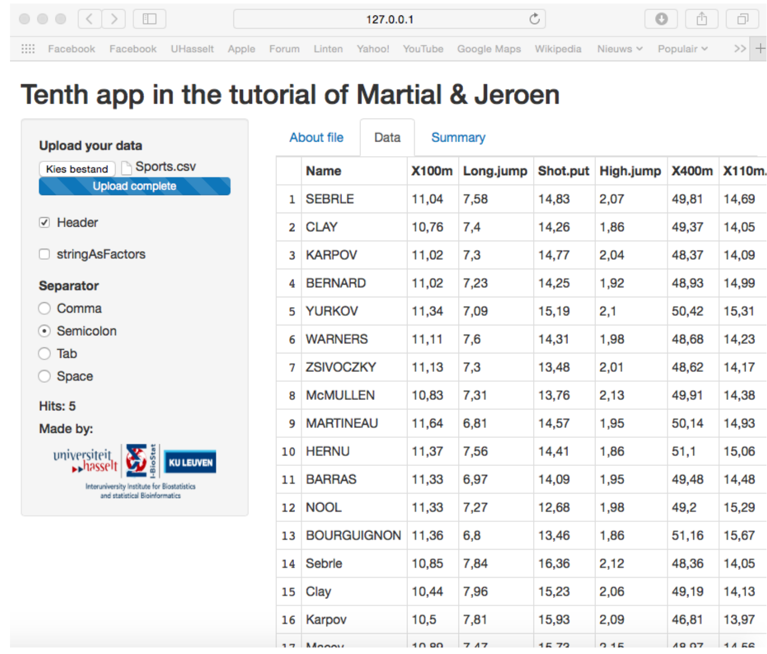
```
> ui <- fluidPage(
        titlePanel(title="Ninth app ..."),
        sidebarLayout(
        sidebarPanel(
         fileInput(...),
         checkboxInput(...),
         checkboxInput(...),
         radioButtons(...),
         h5(...),
         div(tags$img(...)),
        mainPanel(
         uiOutput("tb")
        )))
> server <- function(input, output){
      data <- reactive({...})
      output$filedf <- renderTable({...})
      output$sum <- renderTable({...})
      output$table <- renderTable({...})
      output$tb <- renderUI({...})
}
> shinyApp(ui = ui, server = server)
```



- **App 10:**
  Extend **App 9** by adding a visitor hit counter in the slider panel.

  Display:

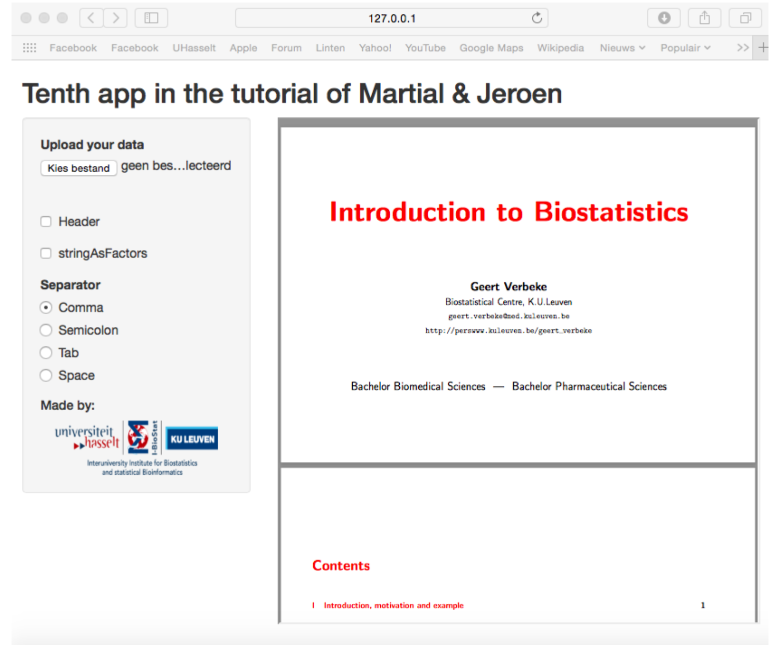




	Name	X100m	Long.jump	Shot.put	High.jump	X400m	X110m.
1	SEBRLE	11,04	7,58	14,83	2,07	49,81	14,69
2	CLAY	10,76	7,4	14,26	1,86	49,37	14,05
3	KARPOV	11,02	7,3	14,77	2,04	48,37	14,09
4	BERNARD	11,02	7,23	14,25	1,92	48,93	14,99
5	YURKOV	11,34	7,09	15,19	2,1	50,42	15,31
6	WARNERS	11,11	7,6	14,31	1,98	48,68	14,23
7	ZSIVOCZKY	11,13	7,3	13,48	2,01	48,62	14,17
8	McMULLEN	10,83	7,31	13,76	2,13	49,91	14,38
9	MARTINEAU	11,64	6,81	14,57	1,95	50,14	14,93
10	HERNU	11,37	7,56	14,41	1,86	51,1	15,06
11	BARRAS	11,33	6,97	14,09	1,95	49,48	14,48
12	NOOL	11,33	7,27	12,68	1,98	49,2	15,29
13	BOURGUIGNON	11,36	6,8	13,46	1,86	51,16	15,67
14	Sebrle	10,85	7,84	16,36	2,12	48,36	14,05
15	Clay	10,44	7,96	15,23	2,06	49,19	14,13
16	Karpov	10,5	7,81	15,93	2,09	46,81	13,97

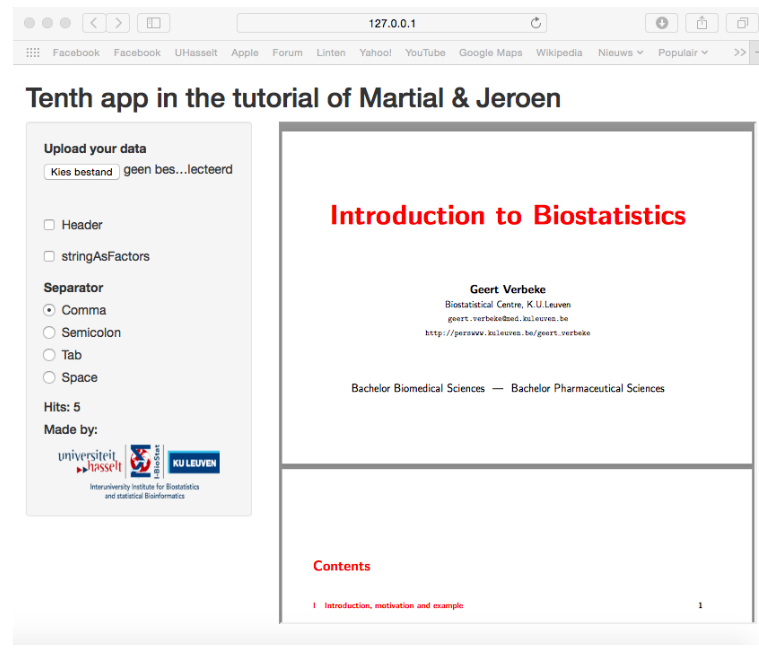
Step 1: Maintain coding structure of **App 9**

```
> ui <- fluidPage(
        titlePanel(title="Tenth app ..."),
        sidebarLayout(
        sidebarPanel(
         fileInput(...),
         checkboxInput(...),
         checkboxInput(...),
         radioButtons(...),
         h5(...),
         div(tags$img(...)),
        mainPanel(...)))
> server <- function(input , output){
      data <- reactive({...})
      output$filedf <- renderTable({...})
      output$sum <- renderTable({...})
      output$table <- renderTable({...})
      output$tb <- renderUI({...})
}
> shinyApp(ui = ui , server = server)
```



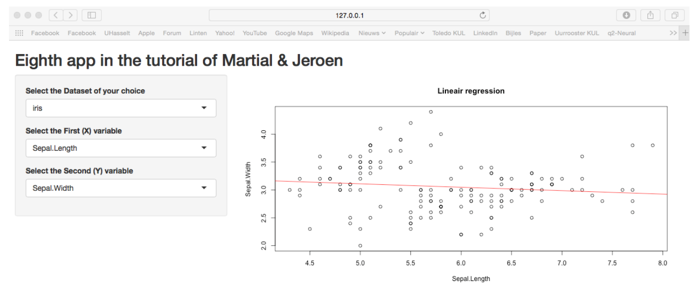
Step 2: Create function in R that counts visitors & link them to UI

```
> ui <- fluidPage(
        titlePanel(title="Tenth app ..."),
        sidebarLayout(
        sidebarPanel(
         fileInput(...),checkboxInput(...),
         checkboxInput(...),radioButtons(...),
         h5(textOutput("counter")),
         h5(...),
         div(tags$img(...)),
        mainPanel(...)))
> server <- function(input , output){
     data <- reactive({...})
     output$filedf <- renderTable({...})
     output$sum <- renderTable({...})
     output$table <- renderTable({...})
     output$tb <- renderUI({...})
     output$counter <- renderText({
       if(!file.exists("counter.Rdata"))
        counter <- 0
       else
         load(file="counter.Rdata")
       counter <- counter + 1
       save(counter, file="counter.Rdata")
       paste("Hits: ", counter)
       })
}
> shinyApp(ui = ui , server = server)
```



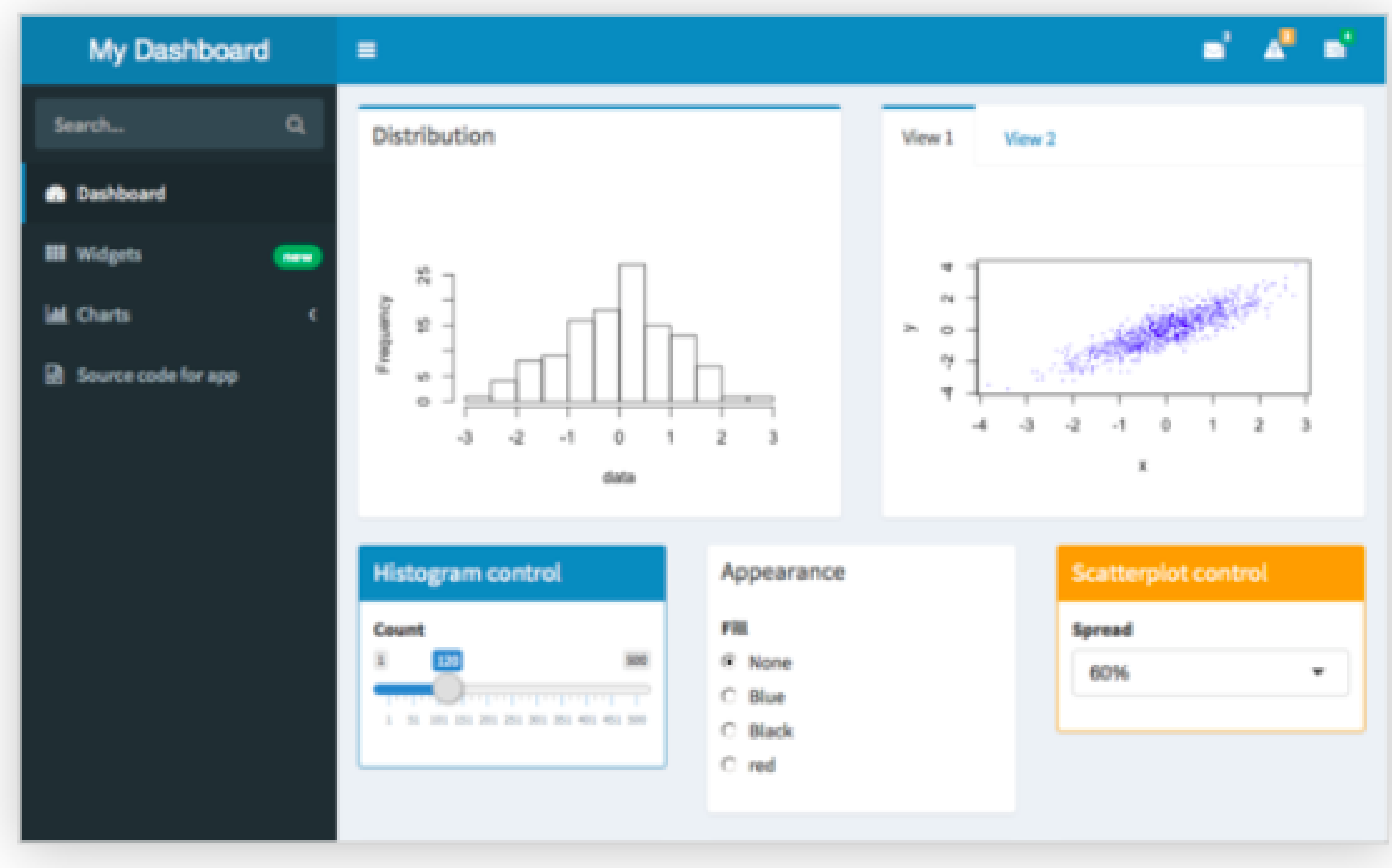
3.4 Extension to dashboard shells

- Build-in UI framework of Shiny is simple of nature



- More 'fancy' layouts/dashboards are often preferred by the user, which can be done with HTML, CSS and Javascript widgets
    - **Advantage:** Own layout structures can be developed
    - **Disadvantage:** Time consuming to programme

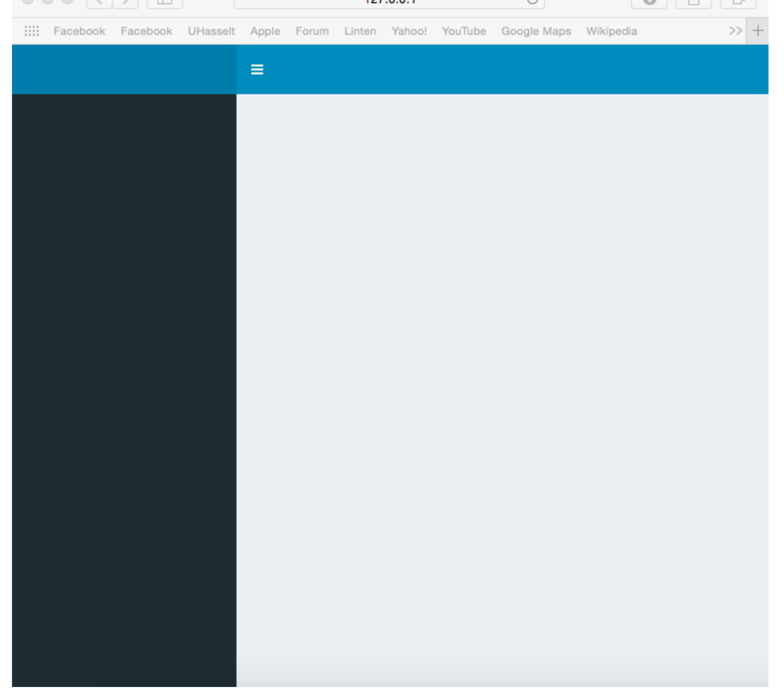
- **Alternative way:** Make use of the **shinydashboard** package, which makes it easy to use **Shiny** and create dashboards like these:



- A Shiny dashboard has three parts, i.e., a **header**, a **sidebar**, and a **body**, and is build up with the following template in R:

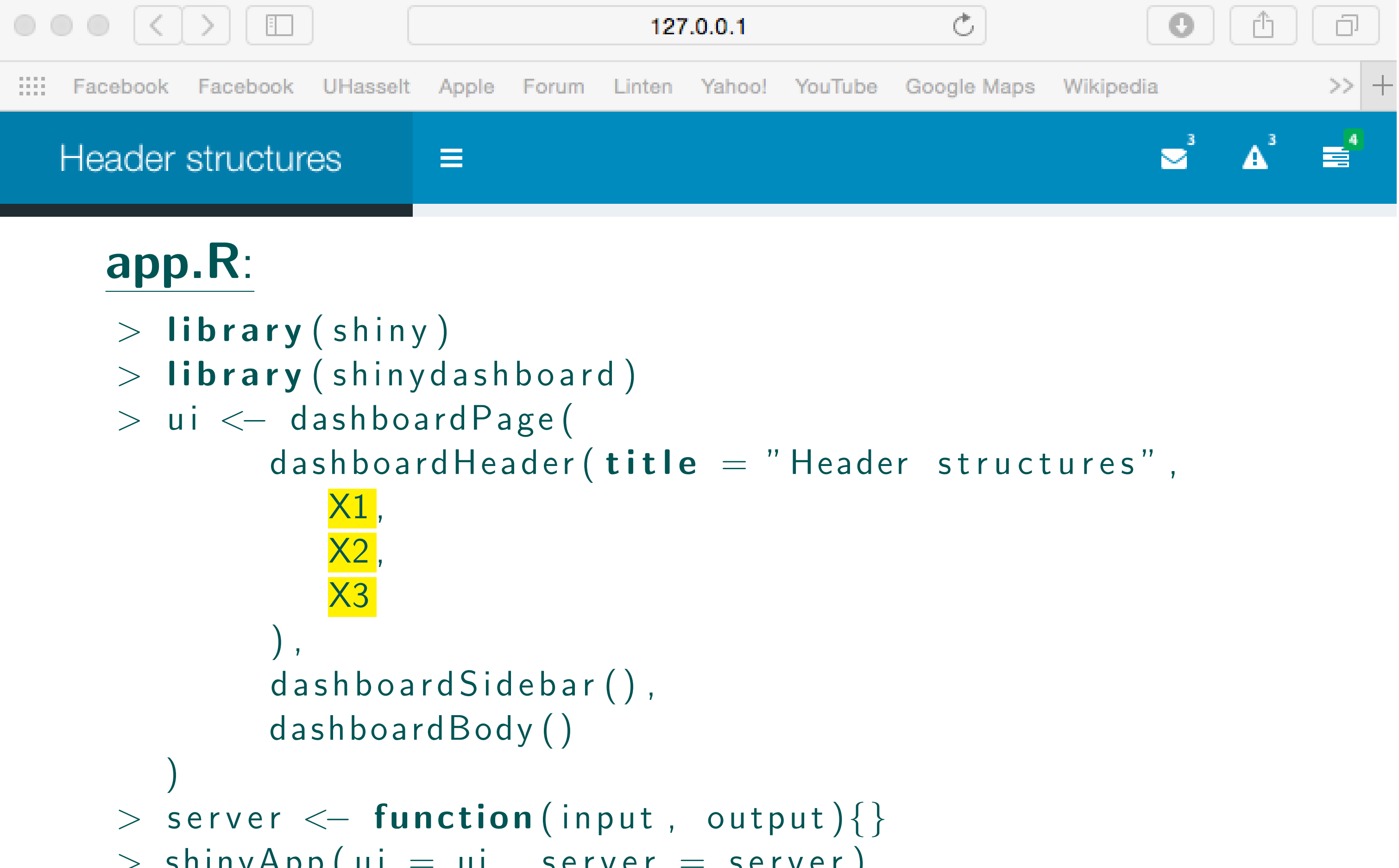
app.R:

```
> library(shiny)
> library(shinydashboard)
> ui <- dashboardPage(
      dashboardHeader(),
      dashboardSidebar(),
      dashboardBody()
      )
> server <- function(input, output){}
> shinyApp(ui = ui, server = server)
```



- Next to standard shiny statements/functions/etc. (Chapter 2), extra features can be added in the dashboard framework:

1. **Header:**

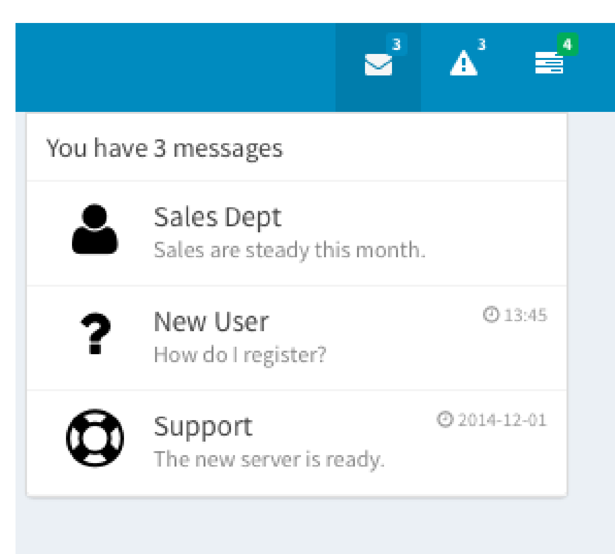


app.R:

```
> library(shiny)
> library(shinydashboard)
> ui <- dashboardPage(
        dashboardHeader(title = "Header structures",
            X1,
            X2,
            X3
        ),
        dashboardSidebar(),
        dashboardBody()
  )
> server <- function(input, output){}
> shinyApp(ui = ui, server = server)
```

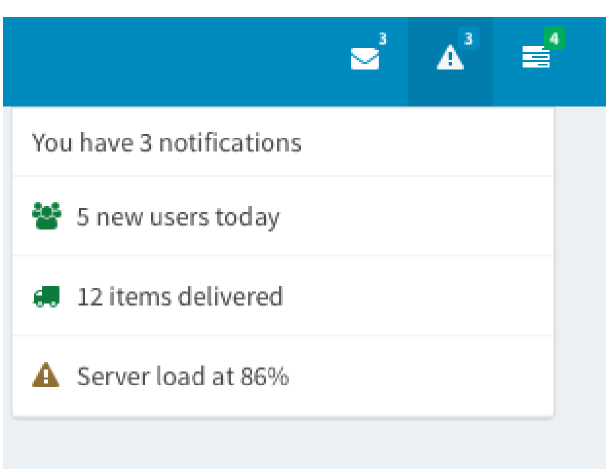
- Message menus (X1)

```
dropdownMenu(type = "messages",
  messageItem(
    from = "Sales Dept",
    message = "Sales are steady this month."
  ),
  messageItem(
    from = "New User",
    message = "How do I register?",
    icon = icon("question"),
    time = "13:45"
  ),
  messageItem(
    from = "Support",
    message = "The new server is ready.",
    icon = icon("life-ring"),
    time = "2014-12-01"
  )
)
```



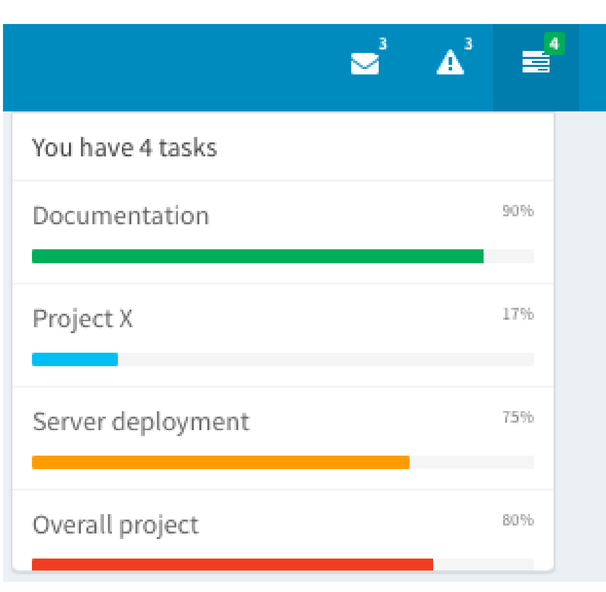
- Notification menus (X2)

```
dropdownMenu(type = "notifications",
  notificationItem(
    text = "5 new users today",
    icon("users")
  ),
  notificationItem(
    text = "12 items delivered",
    icon("truck"),
    status = "success"
  ),
  notificationItem(
    text = "Server load at 86%",
    icon = icon("exclamation-triangle"),
    status = "warning"
  )
)
```



- Task menus (X3)

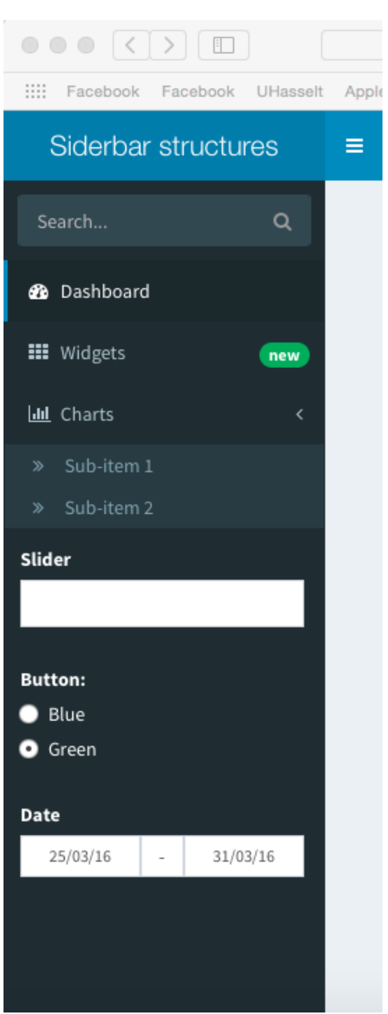
```
dropdownMenu(type = "tasks",
        badgeStatus = "success",
  taskItem(value = 90, color = "green",
    "Documentation"
  ),
  taskItem(value = 17, color = "aqua",
    "Project X"
  ),
  taskItem(value = 75, color = "yellow",
    "Server deployment"
  ),
  taskItem(value = 80, color = "red",
    "Overall project"
  )
)
```



2. **Sidebar:**

app.R:

```
> ui <- dashboardPage(
        dashboardHeader(title = "Siderbar ..."),
        dashboardSidebar(
            X1,
            X2,
            X3
        ),
        dashboardBody()
    )

> server <- function(input, output){}

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



- Sidebar search form (X1)

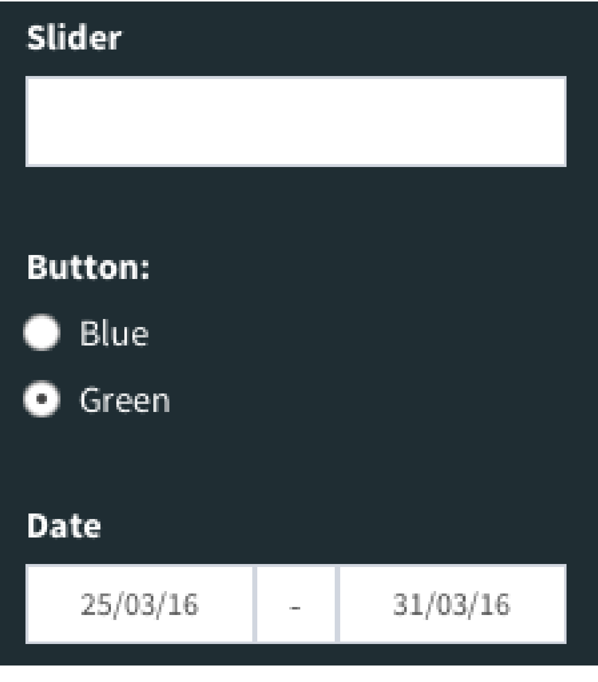
```
sidebarSearchForm(textId = "searchText",
        buttonId = "searchButton",
        label = "Search...")
```

- Menu items and tabs (X2)

```
sidebarMenu(
  menuItem("Dashboard", tabName = "dashboard",
      icon = icon("dashboard")),
  menuItem("Widgets", icon = icon("th"),
      tabName = "widgets", badgeLabel = "new",
      badgeColor = "green"),
  menuItem("Charts", icon=icon("bar-chart-o"),
      menuSubItem("Sub-item 1",
        tabName = "subitem1"),
      menuSubItem("Sub-item 2",
        tabName = "subitem2")
   )
)
```

- Traditional inputs (X3)

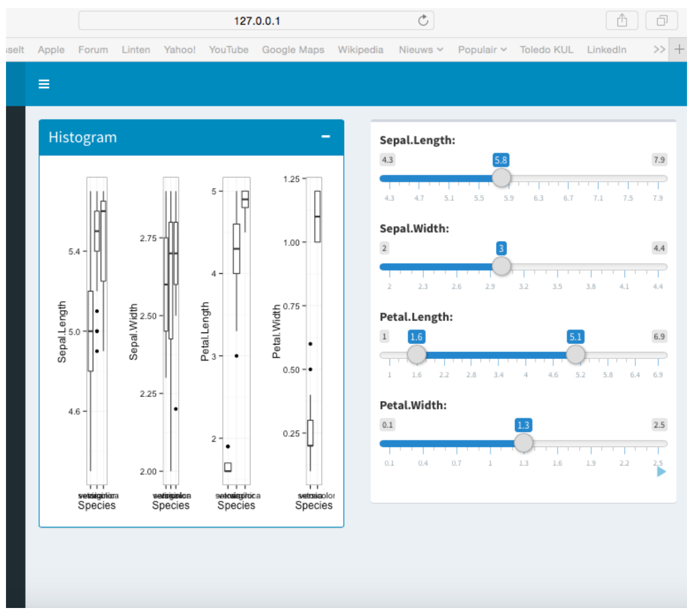
```
textInput(inputId="slider", label="Slider"),
radioButtons(inputId="color", "Button:",
  list("Blue","Green"),"Green"),
dateRangeInput('dateRange2',label = "Date",
  start=Sys.Date()-3, end=Sys.Date()+3,
  min=Sys.Date()-10, max=Sys.Date()+10,
  separator="-", format="dd/mm/yy",
  startview='year', language='fr',
  weekstart = 1)
```



3. **Body:**

app.R:

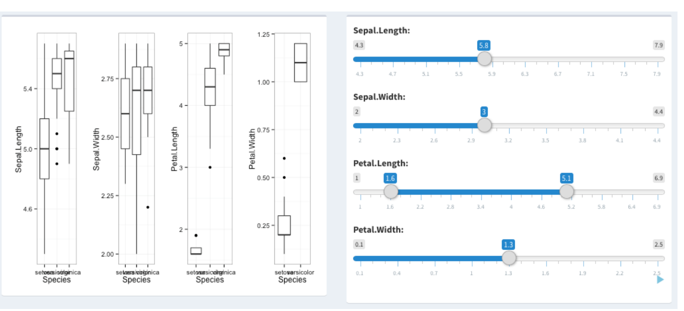
```
> ui <- dashboardPage(
        dashboardHeader(title="Siderbar
            ..."),
        dashboardSidebar(),
        dashboardBody(
            X1
        )
   )

> server <- function(input, output){...}

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



- Boxes (X1)

Example 1:

```
fluidRow(
  box(
    plotOutput(outputId = "box")),
  box(
    sliderInput(inputId="sepallength",
      "Sepal.Length:", min=4.3, max=7.9,
      value=5.8, step=0.1),
    sliderInput(inputId="sepalwidth",
      "Sepal.Width:", min=2, max=4.4,
      value=3, step=0.1),
    sliderInput(inputId="petallength",
      "Petal.Length:", min=1, max=6.9,
      value = c(1.6, 5.1)),
    sliderInput(inputId="petalwidth",
      "Petal.Width:", min=0.1, max = 2.5,
      value = 1.3, step = 0.3, animate=
      animationOptions(interval=2600,
      loop=TRUE))
  )
)
```



Example 2:

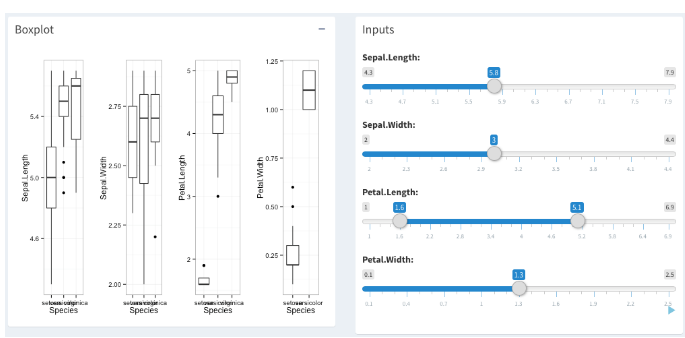
```
fluidRow(
  box(
    title = "Boxplot", status = "primary",
    plotOutput(outputId = "box")),
  box(
    title = "Inputs", status = "warning",
    sliderInput(...), sliderInput(...),
    sliderInput(...), sliderInput(...)
  ))
```

Example 3:

```
fluidRow(
  box(
    title = "Boxplot", status = "primary",
    solidHeader=TRUE, collapsible=TRUE,
    plotOutput(outputId = "box")),
  box(
    title = "Inputs", status = "warning",
    solidHeader = TRUE,
    sliderInput(...), sliderInput(...),
    sliderInput(...), sliderInput(...)
  ))
```

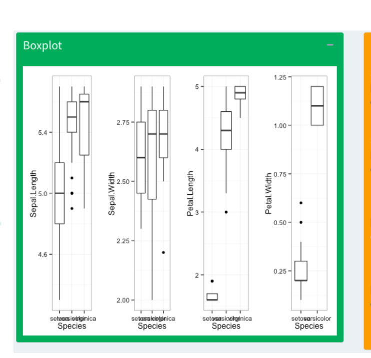
Example 4:

```
fluidRow(
  box(
    title = "Boxplot", solidHeader=TRUE,
    collapsible=TRUE, plotOutput(...)),
  box(
    title = "Inputs", solidHeader = TRUE,
    sliderInput(...), sliderInput(...),
    sliderInput(...), sliderInput(...)
  )
)
```



Example 5:

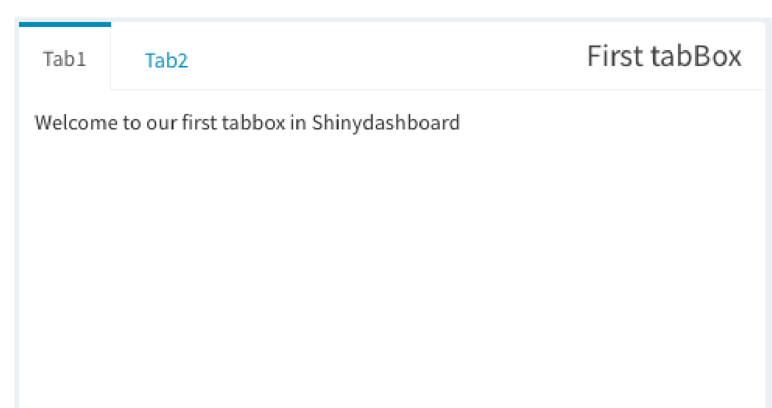
```
fluidRow(
  box(
    title = "Boxplot", background="green",
    solidHeader=TRUE, plotOutput(...)),
  box(
    title = "Inputs", background="yellow",
    sliderInput(...), sliderInput(...),
    sliderInput(...), sliderInput(...)
  )
)
```



- Tabbox (X1)

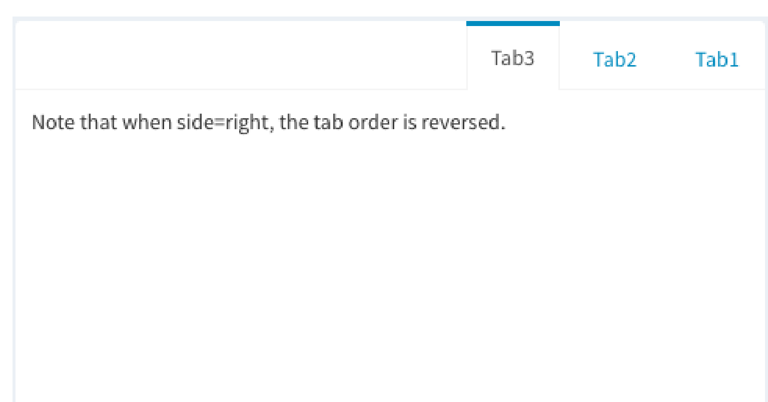
Example 1:

```
tabBox(
  title = "First tabBox",
  id = "tabset1", height = "250px",
  tabPanel("Tab1", "Welcome ..."),
  tabPanel("Tab2", "Tab content 2")
)
```



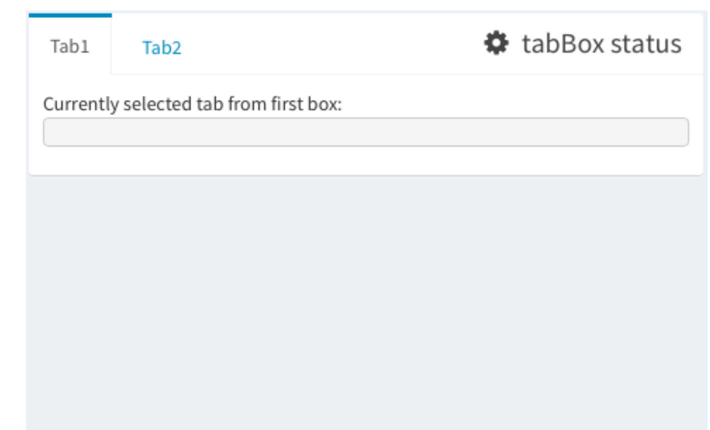
Example 2:

```
tabBox(
  side = "right", height = "250px",
  selected = "Tab3",
  tabPanel("Tab1", "Tab content 1"),
  tabPanel("Tab2", "Tab content 2"),
  tabPanel("Tab3", "Note that ...")
)
```

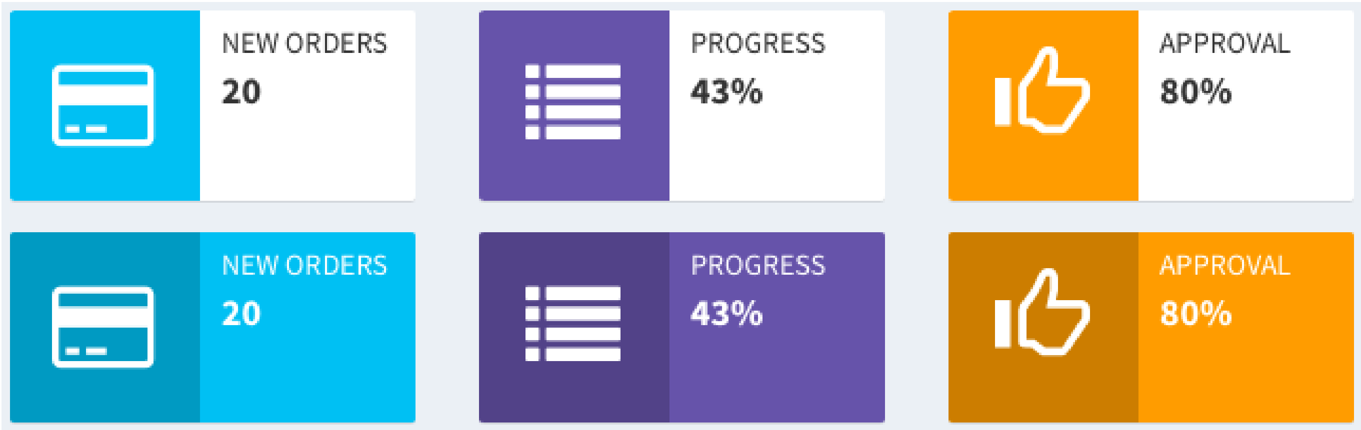


Example 3:

```
tabBox(
  title = tagList(shiny::icon("gear"),
    "tabBox status"),
  tabPanel("Tab1",
    "Currently ...:",
    verbatimTextOutput("tabset1Selected")
  ),
  tabPanel("Tab2", "Tab content 2")
)
```

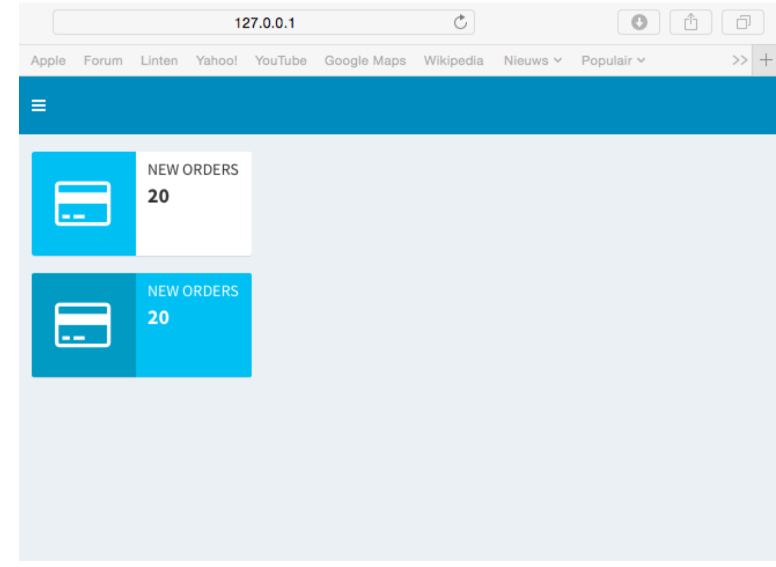


- Infobox



Step 1: Input two different infoboxes in the dashboard body

```
> library(shinydashboard)
> ui <- dashboardPage(
    dashboardHeader(title="..."),
    dashboardSidebar(),
    dashboardBody(
      fluidRow(
       infoBox("New Orders", 10*2,
        icon=icon("credit-card"))),
      fluidRow(
       infoBox("New Orders", 10*2,
        icon=icon("credit-card"),fill
        =TRUE))
    )
  )
> server <- function(input , output){}
> shinyApp(ui = ui , server = server)
```



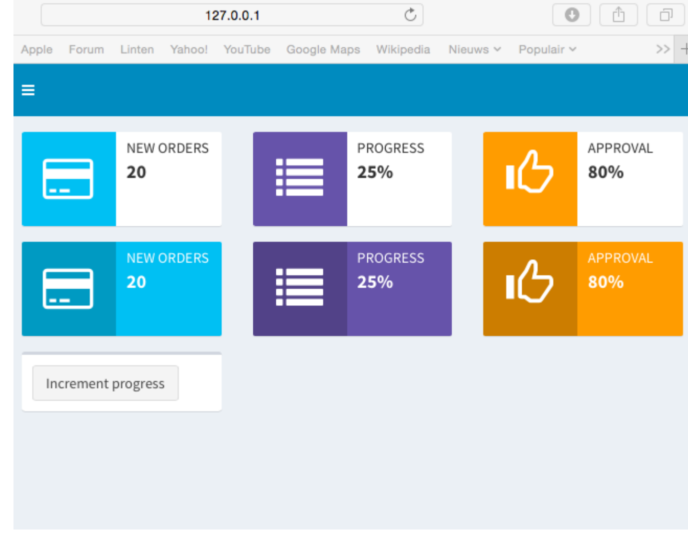
Step 2: Use renderInfobox() in R

```
> ui <- dashboardPage(...)
> server <- function(input , output){
    output$progrBox<-renderInfoBox({
     infoBox(
      "Progress",paste0(25+input$count,
      "%"),icon=icon("list"),color=
      "purple")})
    output$apprBox<-renderInfoBox({
     infoBox(
      "Approval","80%",icon=
      icon("thumbs-up",lib="glyphicon"),
      color="yellow")})
    output$progrBox2<-renderInfoBox({
     infoBox(
      "Progress",paste0(25+input$count,
      "%"),icon=icon("list"),color=
      "purple",fill=TRUE)})
    output$apprBox2<-renderInfoBox({
     infoBox(
      "Approval","80%",icon=
      icon("thumbs-up",lib="glyphicon"),
      color="yellow",fill=TRUE)})}
> shinyApp(ui = ui , server = server)
```

127.0.0.1
Apple Forum Linten Yahoo! YouTube Google Maps Wikipedia Nieuws Populair
NEW ORDERS
20
NEW ORDERS
20

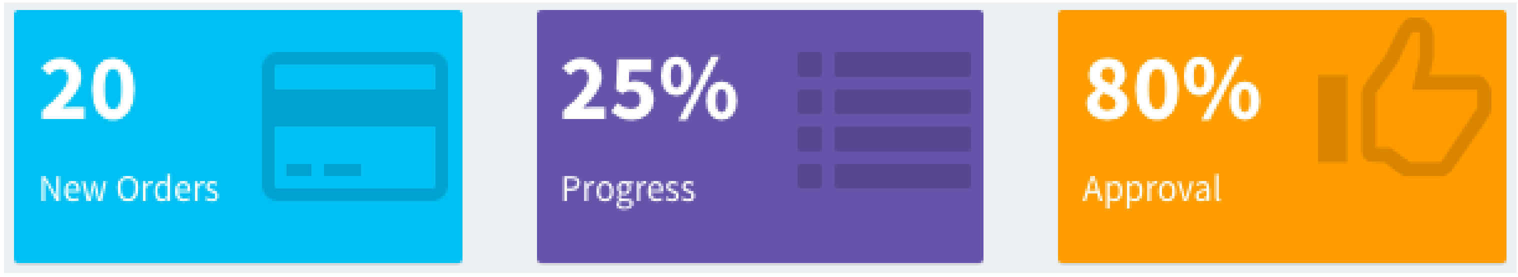
Step 3: Link renderInfobox() to UI with infoBoxOutput()

```
> ui <- dashboardPage(
    dashboardHeader(title="..."),
    dashboardSidebar(),
    dashboardBody(
     fluidRow(...,
      infoBoxOutput("progrBox"),
      infoBoxOutput("apprBox")
     ),
     fluidRow(...,
      infoBoxOutput("progrBox2"),
      infoBoxOutput("apprBox2")
     ),
     fluidRow(
      box(width = 4,
       actionButton("count",
        "Increment progress")))
    )
   )
> server <- function(input , output){
   output$progrBox<-renderInfoBox({...})
   output$apprBox<-renderInfoBox({...})
   output$progrBox2<-renderInfoBox({...})
   output$apprBox2<-renderInfoBox({...})}
> shinyApp(ui = ui , server = server)
```



- Valuebox

  → Similar to the Infobox (i.e., **infoBox(...)** ⇒ **valueBox(...)**, etc.)

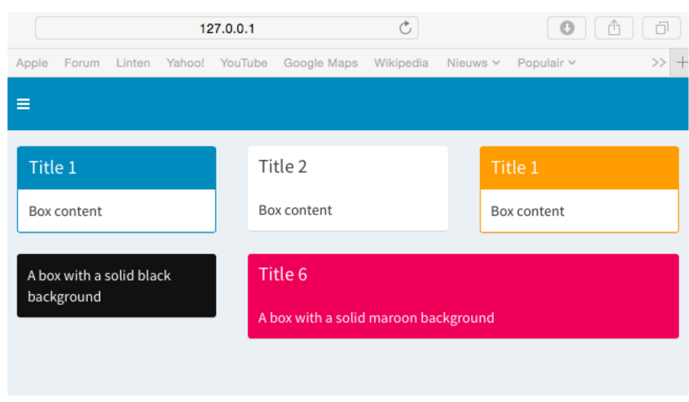


4. **Layout:**

- Row-based

app.R:

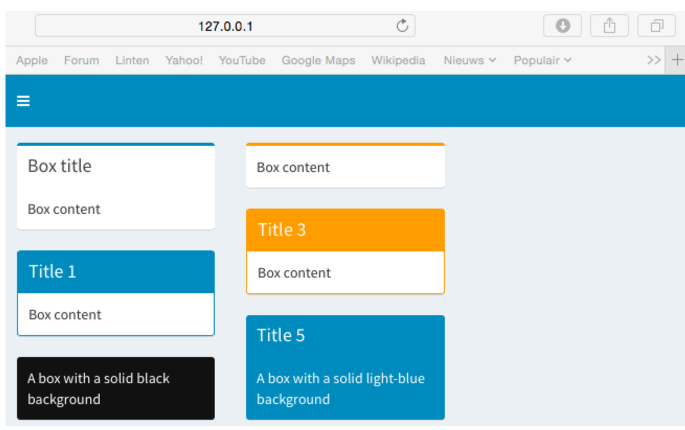
```
> ui <- dashboardPage(
        dashboardHeader(title="..."),
        dashboardSidebar(),
        dashboardBody(
            fluidRow(...),
            fluidRow(...)
        )
    )

> server <- function(input, output){...}

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



- Column-based

app.R:

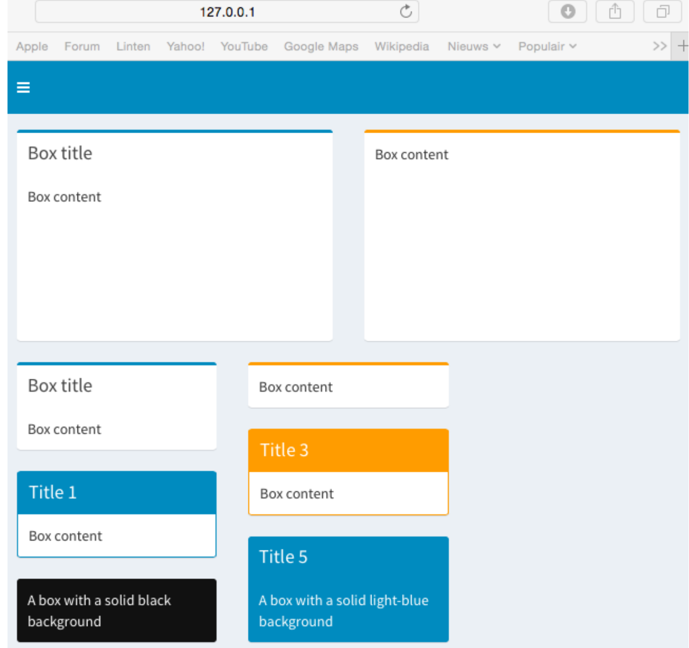
```
> ui <- dashboardPage(
      dashboardHeader(title="..."),
      dashboardSidebar(),
      dashboardBody(
        fluidRow(
          column(width = 4,
            box(...),
            box(...),
            box(...)),
          column(width = 4,
            box(...),
            box(...),
            box(...))
        )
      )
  )

> server <- function(input, output){...}

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



- Mixed row and column layout

app.R:

```
> ui <- dashboardPage(
        dashboardHeader(title="..."),
        dashboardSidebar(),
        dashboardBody(
            fluidRow(...) ,
            fluidRow(
             column(width = 4,
               box(...) ,
               box(...) ,
               box(...)) ,
             column(width = 4,
               box(...) ,
               box(...) ,
               box(...))
            )
        )
    )

> server <- function(input, output){...}

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

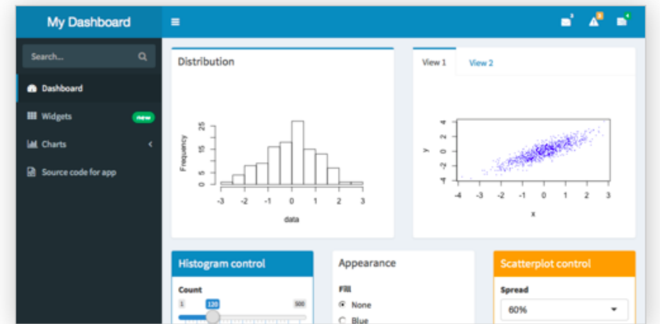


5. **Color themes:**

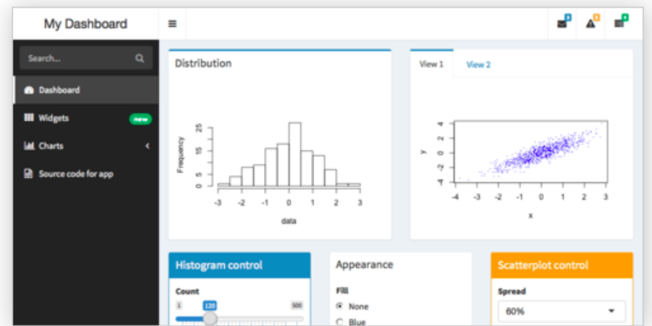
app.R:

```
> ui <- dashboardPage(skin="X1",
        dashboardHeader(title="..."),
        dashboardSidebar(...),
        dashboardBody(...)
      )

> server <- function(input, output){...}

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

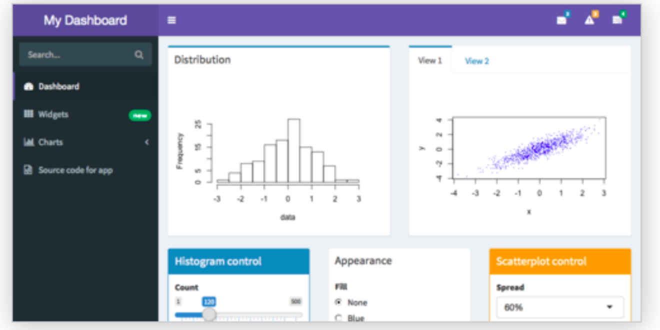
- Blue (X1)



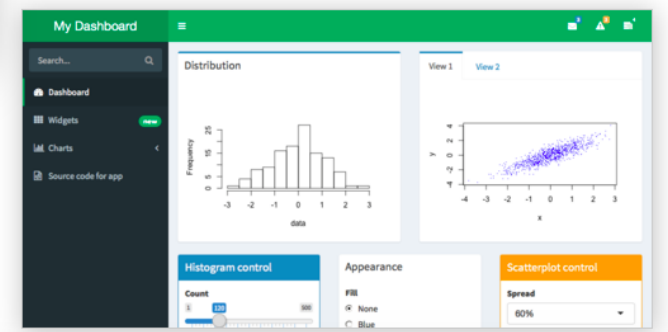
- Black (X1)



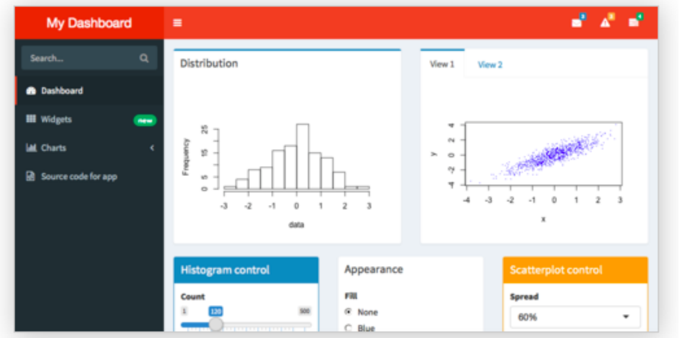
- Purple (X1)



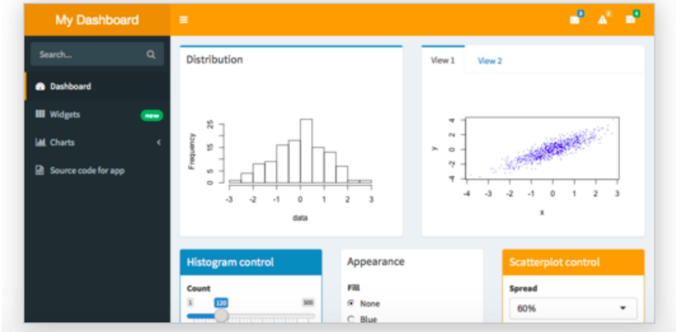
- Green (X1)



- Red (X1)



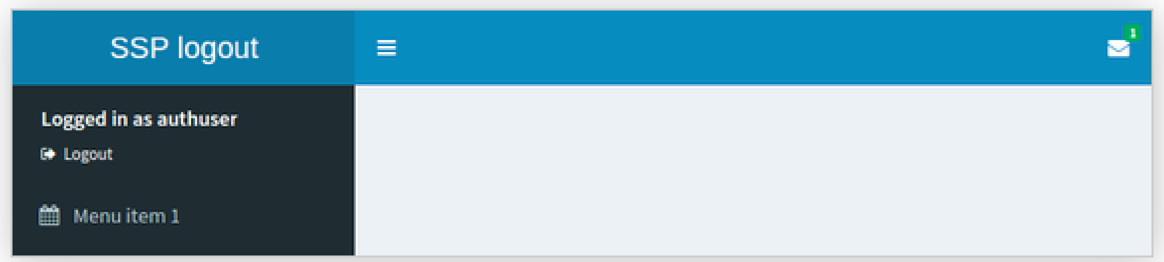
- Yellow (X1)



6. **Extras:**

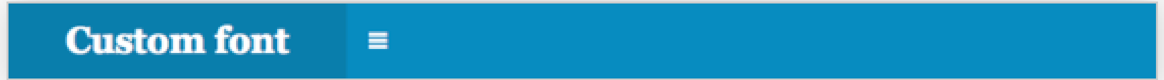
- Logout panel

  Usage: Deploying with Shiny Server Pro (SSP)



- CSS

  Usage: Customizing font



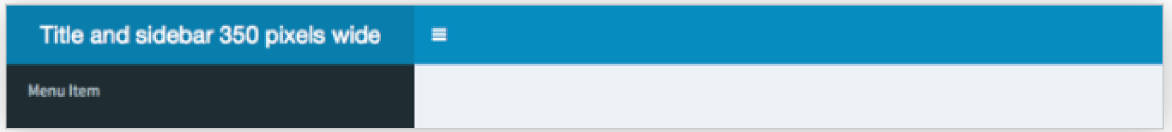
- Long-titles

  Code: titleWidth = 450

Example of a long title that needs more space

- Sidebar-width

  Code: width = 450



- Icons
  - ▷ **Font-Awesome**

    URL: *http://fontawesome.io/icons/*

    Code: icon("calendar") (by default)
  - ▷ **Glyphicons**

    URL: *http://getbootstrap.com/components/#glyphicons*

    Code: icon("cog", lib = "glyphicon")

Calendar from Font-Awesome:

Cog from Glyphicons:

- Statuses and colors
  - **Statuses**

    Code: status = "primary"

  - **Color**

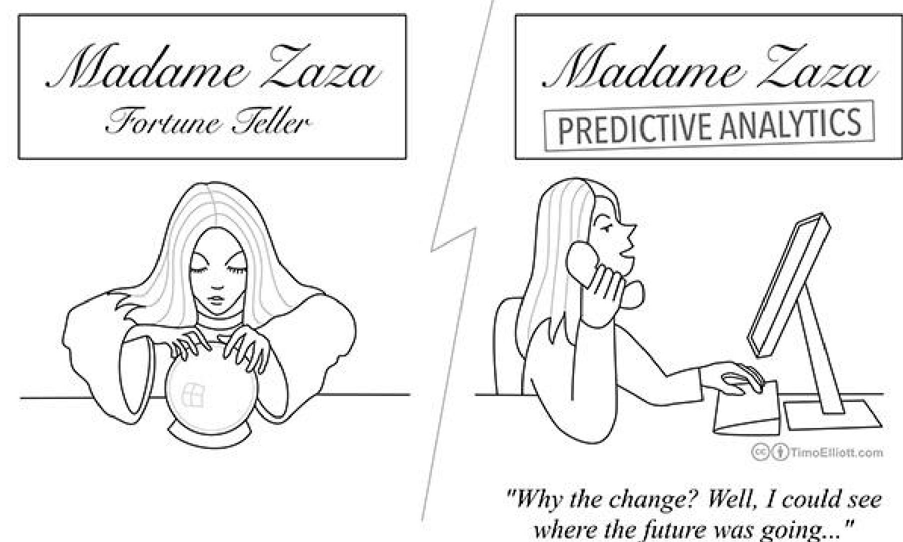
    Code: color="red"

Part 4:

Applications in the area of biostatistics & data science

Chapter 4:
Apps in biostatistics & data science

- Shiny as an extra tool ...
- Other way of constructing Shiny apps
- To the applications ...

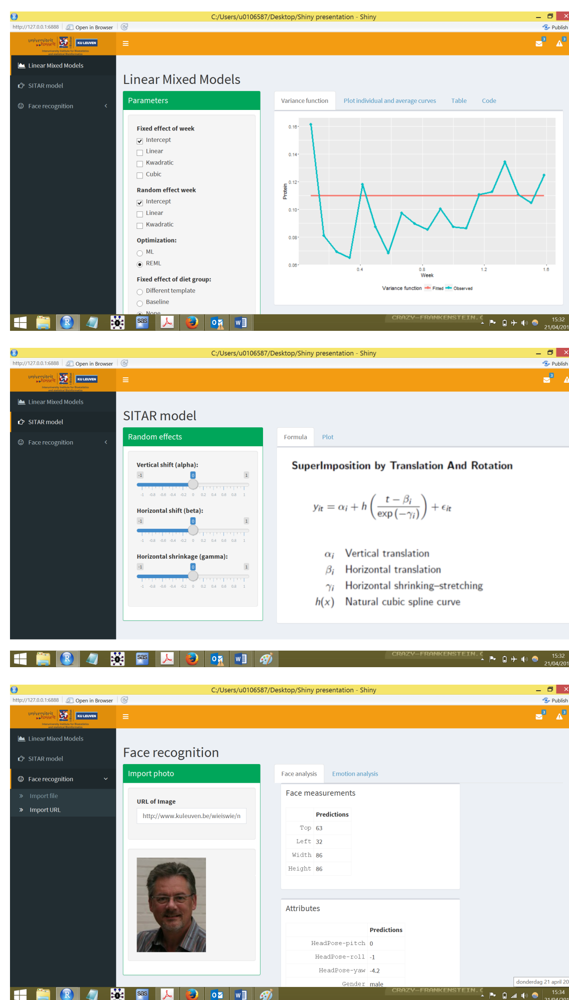


4.1 Shiny as an extra tool ...

1. Teaching purposes
   (e.g., Linear Mixed Models app)

2. Visualizing complex models
   (e.g. Sitar Model app)

3. As a nice layout
   (e.g., Face Recognition app)



4.2 Other way of constructing Shiny apps

```
> dataset <- read.delim()
> Function1 <- function(par1, par2){
        return(list(obj1, obj2))
   }
> Function2 <- function(par3){
        Return(list(obj3, obj4))
   }

> ui <- (
        plotOutput('oObj1') ... tableOutput('oObj2') ...
        numericInput('iPar1', ...)
   )

> server <- function(input, output){
        output$oObj1 <- renderPlot(Function1(input$iPar1, input$iPar2)[[1]])
        output$oObj2 <- renderTable(Function1(input$iPar1, input$iPar2)[[2]])
        output$oObj3 <- renderText(Function2(input$iPar3)[[1]])
        output$oObj4 <- renderImage(Function2(input$iPar3)[[2]])
  }
```

4.3 To the applications ...

References

Beeley, C. (2013). Web Application Development with R Using Shiny. *Packt Publishing*, **110 pages**

Chang, W. (2013). R Graphics Cookbook (1st ed.). *O'Reilly Media*, **416 pages**

Teetor, P. (2011). Wickham, H. (2010). R Cookbook (O'Reilly Cookbooks) (1st ed.). *O'Reilly Media*, **438 pages**

Wickham, H. (2010). ggplot2: Elegant Graphics for Data Analysis (Use R!) (3rd ed.). *Springer*, **213 pages**