Statistical programming using R

Lecture 6 – Interactive Dashboards using `Shiny`

Introduction

You have spent a lot of time learning programming and data analysis. You know how to load data, clean data, analyse and do data visualisation. You have produced impressive reports for different research projects. All your reports were static documents submitted as MS Word or PDF files. This is perfectly fine if your data do not change too quickly. But what if data change? For example, you get new data every day. Alternative problem is changing some conditions of the report. For example, your reader wants to try more or less clusters than in your report. Should you re-do the entire report?

To solve all above problems, you need an interactive report, live report that can change itself if something changes – data or input parameters. In R there is a fantastic tool for that – package `shiny`. It allows to create a dashboard that can be opened as a web page with R code running behind it.

You can find a detailed introduction into `shiny` here – https://shiny.rstudio.com/tutorial/ . This tutorial is highly recommended. Our lecture will repeat some ideas from that tutorial but also provides extra information that should support your work on the project.

Obviously, there is a web site dedicated to `shiny` – https://shiny.rstudio.com/ , where you can find a cheat sheet – https://shiny.rstudio.com/images/shiny-cheatsheet.pdf ; and a reference for your version of `shiny` – https://shiny.rstudio.com/reference/shiny/ – where all functions are nicely organised by their functional groups.

The basic structure of a shiny dashboard or shiny app is very simple:

```
# load the library for shiny and also any other library you need
library(shiny)

# part to create a user interface
# it has two parts - inputs and outputs
ui <- fluidPage(
  # inputs,
  # outputs
)

# part to do data analysis based on input parameters and expected outputs
server <- function(input, output) {
  # normal R code with 3 special rules
}

# part to combine above two parts and create an app, no need to change it
shinyApp(ui = ui, server = server)
```

The task of creating an interactive dashboard consists of two steps:

1. Design a user interface (UI): decide what types of input arguments you need, decide what types of outputs you want and decide how to place them on the page.

2. Create R code that does data analysis by taking parameters stored inside `input` variable and storing appropriate results in `output` variable. This is important to match formats of the output pre-defined inside UI and loaded inside server function.

User Interface

User interface part of the code above is a function itself. Its purpose is to create an HTML page with all input and output objects. There are multiple different functions that can be used for that. You will see examples later. Also, check the reference section "UI Layout". Function `fluidPage()` is good for many types of dashboards, it creates a fluid layout that fits nicely in any web browser.

Arguments for user interface functions are input and output functions in the desired order and separated by commas as any arguments in any function.

Input functions

There is a number of input functions that create UI and store values for `shiny` app input arguments.

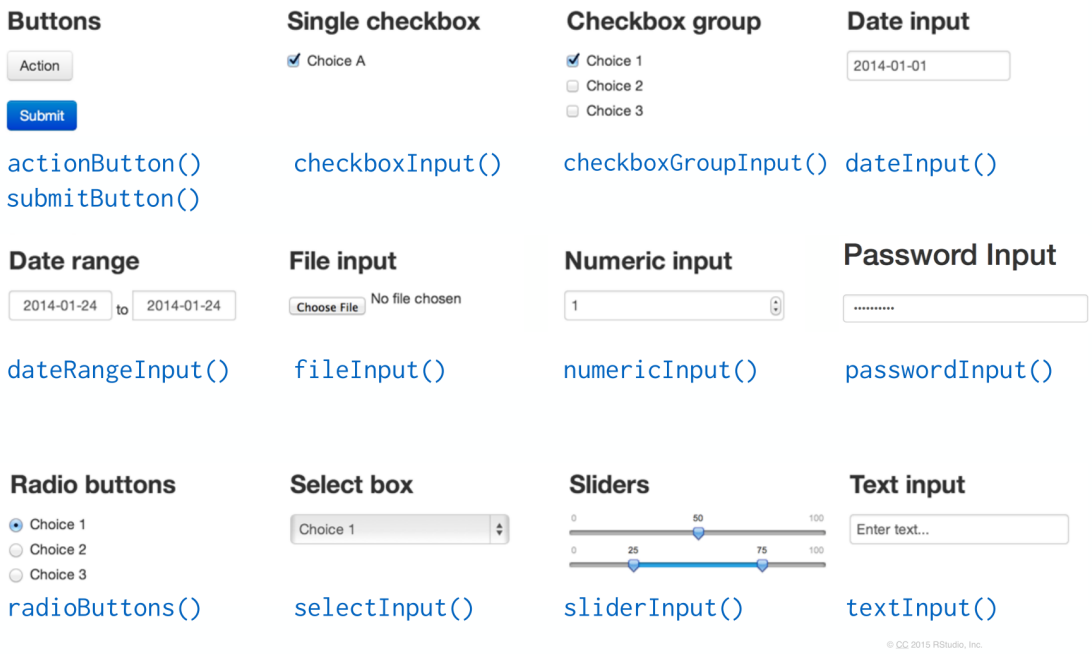


Figure 1: Input functions

To see details of each function check the reference guide section "UI Inputs". Also, have a look on the gallery of widgets with examples of all input functions – https://shiny.rstudio.com/gallery/widget-gallery.html

Each input function has common arguments and unique arguments. Arguments common for all input functions are `inputId` and `label`. The first one is used to identify each input object later in the server function. Obviously, id should be unique for each input function. The second one – `label` – is used to show the title of the object on the dashboard. It can be absolutely anything but you want it to be something easy to understand for the user of the dashboard.

Unique arguments of the input function are different for each function. Some unique arguments are compulsory but majority of them are optional, so you don't really need to change them (however, you can if you want/need to).

```r
# example of input function with common arguments (first two) and
# unique compulsory arguments (last three)
sliderInput(inputId = "num", label = "Choose a number", min = 0, max = 100, value = 50)
```

Output functions

Output functions create objects that visualise information on the dashboard web page. Think about text, numbers, tables, graphs.

Function	Inserts
dataTableOutput()	an interactive table
htmlOutput()	raw HTML
imageOutput()	image
plotOutput()	plot
tableOutput()	table
textOutput()	text
uiOutput()	a Shiny UI element
verbatimTextOutput()	text

Figure 2: Output functions

This list is not complete – all output functions available for your version of `shiny` are listed in the function reference in section UI Outputs.

The only compulsory argument for any output function is `outputId` – reference name for that function. You will use that name when to "load" data analysis result in that function. At the same time, there are a number of optional arguments that can be very handy. For example `width` and `height`, which are self-explanatory – size of output/visualisation. But also `click` and `dblclick` – what to do when user makes a click or double-click on the output object. Obviously, optional functions are different for different output functions.

```r
# example of output function, the first argument is compulsory and
# common for all output functions, everything else is optional and unique for the function
plotOutput(outputId = "hist", width = "100%")
```

Server function

Function `server()` is an engine of the dashboard app. It creates all required outputs based on provided inputs. While *user interface* section might look a bit alien for you, as it works with HTML, function *server*

is very normal – it contains "normal" R code for "normal" analysis. There might be any functions, you can use any packages. You can take any piece of code your created before and include inside *server* function. You just need to follow three rules that are "special" for *server* function.

Rule 1. Any result you what to output should use the same ID as you defined for the output

```
# output function defined in user interface (UI) uses ID "hist"
# plotOutput(outputId = "hist", width = "100%")

server <- function(input, output) {
  # output object should use the same ID
  output$hist <- # code to produce a histogram

}
```

Rule 2. Any output should be processed/prepared to display by function `render*()`.

```
server <- function(input, output) {
  output$hist <- renderPlot({
    # code to produce a histogram
    # you just copy-paste your R code here
    # multiple lines of code are fine is they are placed inside {...}
  })
}
```

The reason is very simple: you can not output R or ggplot graphs or just a data frame outputs on the web page. You have to convert a graph object in to an image. Functions from `render*()` family do it for you. This is a function for each type of data you might want to output.

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

Figure 3: Render functions

Rule 3. All input values defined in *user interface* section are available as elements of `input` variable by using `$`-notation.

```
# input function defined in UI uses ID "num"
# sliderInput(inputId = "num", label = "Choose a number", min = 0, max = 100, value = 50)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num)) # code to create a histogram of "num" random numbers.
  })
}
```

These input values are special as they are connected to input objects and always have a current value of the input parameter. Each time you change input parameter, you force *server* function to run again for the new value/s of input and update relevant outputs.

An ability to automatically run and update the outputs is called *reactivity*. That is, the dashboard reacts on something. There is a number of advanced rules for reactivity. The very first and the most obvious rule of reactivity is a reaction on a change of input values. We have a separate section on the reactivity later.

Complete dashboard

Now we combine all elements presented above to create a dashboard that takes a number (with the help of a slider) as an input, generates a sample of that size of random standardised normal values and plot a histogram.

```
library(shiny)
ui <- fluidPage( # define a fluid design page
  # define input object/function of type slider; don't miss a comma at the end
  sliderInput(inputId = "num", label = "Choose a number", value = 25, min = 1, max = 1000),

  # define output object/function of type plot
  plotOutput(outputId = "my_histogram")
)
server <- function(input, output) {
  # render plot and place it in output object using right ID
  output$my_histogram <- renderPlot({

    # R code to prepare an output - graph; no commas at the end of line
    my_data <- rnorm(input$num)                                    # generate data
    hist(my_data, main = "Random Normal distribution")             # plot the data
    abline(v = mean(my_data), col = "red", lty = 3, lwd = 3)       # add extra line
  })
}
# function to combine user interface and server and then run the dashboard
# I store result as a variable to create PDF document, you just run the function
# and enjoy an interactive dashboard
appToPlot <- shinyApp(ui = ui, server = server)
```

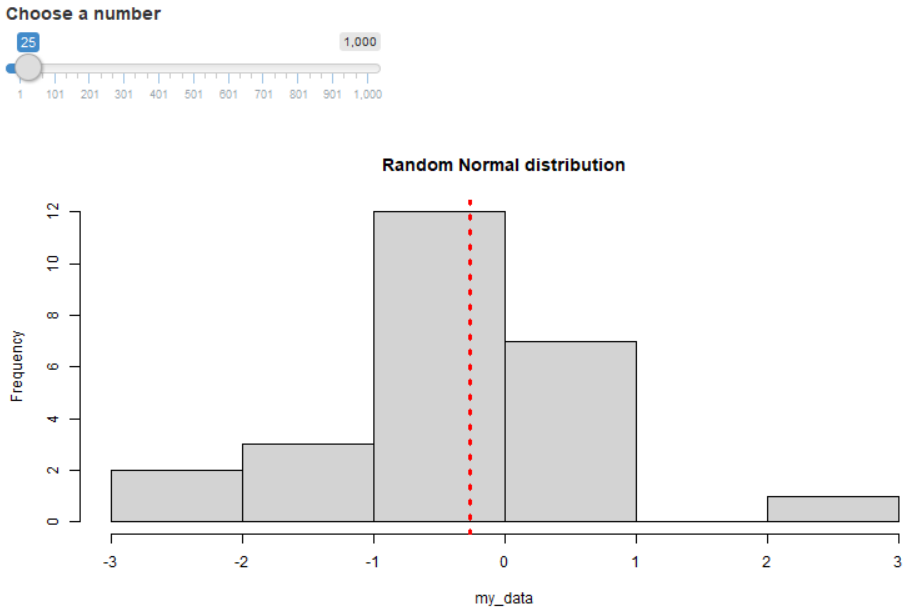


Figure 4: Histogram of normal distribution dashboard

Any type of plotting created by R can be rendered for the dashboard. Below is an example of the same dashboard but with `ggplot2` plotting processed by the custom function.

```r
library(shiny)

# prepare custom function to keep server function tidy
plot_histogram <- function(my_data){
  require(ggplot2)
  my_data <- data.frame(x = my_data)
  ggplot(my_data, aes(x)) + geom_histogram() +
    geom_vline(xintercept = mean(my_data$x), colour = "red", linetype = "dotted", size = 2) +
    ggtitle("Random Normal distribution")
}

# the same user interface function as before
ui <- fluidPage(
  sliderInput(inputId = "num", label = "Choose a number", value = 25, min = 1, max = 1000),
  plotOutput(outputId = "my_histogram")
)

server <- function(input, output) {
  # render plot and place it in output object using right ID
  output$my_histogram <- renderPlot({

    # R code to prepare an output - graph; no commas at the end of line
    my_data <- rnorm(input$num)  # generate data
    plot_histogram(my_data)      # plot the data by custom function prepared
  })
}
# function to combine user interface and server and then run the dashboard
appToPlot <- shinyApp(ui = ui, server = server)
```

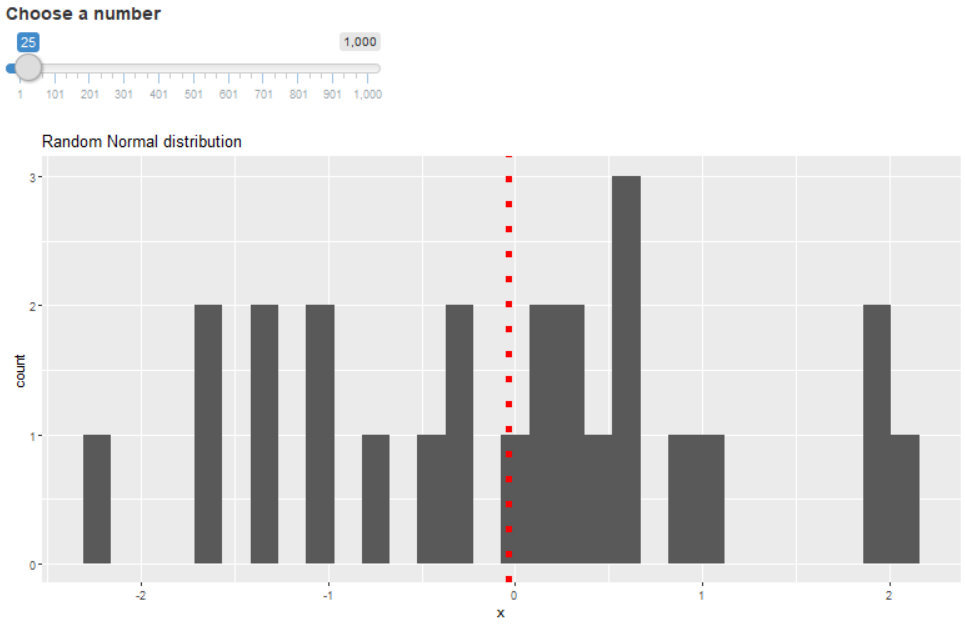
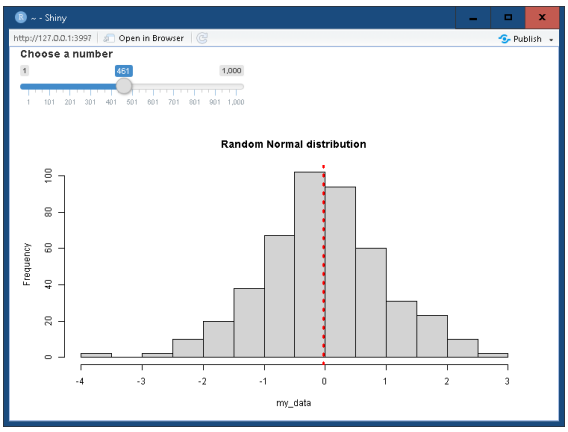


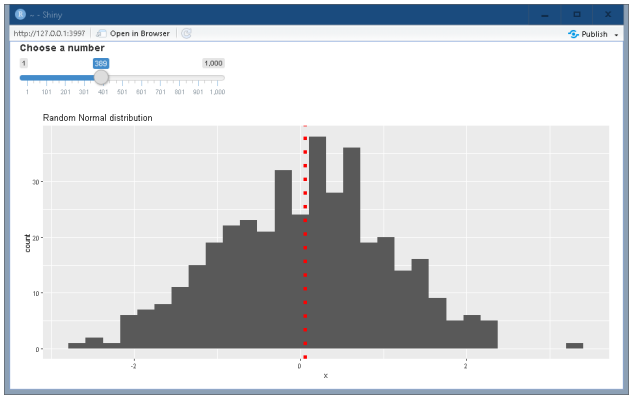
Figure 5: Histogram of normal distribution dashboard

Here it is – your first dashboard! It is interactive and you are ready to share it with the entire world by placing it in the cloud, so it will be available to anyone and at any time. I want to reference you again to

the tutorial – https://shiny.rstudio.com/tutorial/ . It discusses options for hosting shiny app in the cloud. It is not important for us right now.

Below are result dashboards for both versions of the code side by side.





If you need an inspiration on how to make a better dashboard, browse this gallery of shiny apps examples – https://shiny.rstudio.com/gallery/

Reactivity

The main difference between a dashboard and an old-fashion report is an ability of a dashboard to be interactive. Dashboard can do something based on some criteria. The dashboard created in the previous section updates itself in a response for a new information arrived. You change input value – sample size; the dashboard generates a new data set of the given size and plot a histogram for that data set.

Shiny app can do that change due to a *reactivity*. Reactivity is an ability to react in response to a trigger. You've seen one trigger before – change of the input value. However, there can be different triggers. For example:

- a button user needs to click,
- just a click or double-click on the graph,
- text input,
- timer – do something every 30 minutes without any user's input, e.g. download new data.

Reaction on the trigger might be different too, for example:

- update data visualisation,
- run some function,
- change inputs,
- do nothing.

The last option of doing nothing is very important too. We don't always what a reaction or an immediate reaction on our actions.

Reactive values

Reactive values are input values, values that can change at any moment. Shiny app will need to react in response. However, for the reactive values to work, they should be used with reactive functions. There is

an advanced machinery working in every shiny app that we don't see but that makes shiny app interactive. When you change a reactive value, it notifies the function, that uses this value, about the change and reactive function responds. Respond depends on the function, for example `renderPlot()` creates a new graph.

```
# example of reactive value and reactive function
output$my_histogram <- renderPlot({     # renderPlot is reactive function
  my_data <- rnorm(input$num)           # input$num is reactive value
  hist(my_data)
})
```

A reactive value above is used inside a reactive function, hence when `input$num` changes, function `renderPlot()` responds – it "understands" that its "old" output (based on the old input value) is invalid now and function runs again with a new value of `input$num`.

```
# example of incorrect use of reactive value
my_data <- rnorm(input$num)             # input$num is reactive value
output$my_histogram <- renderPlot({     # renderPlot is reactive function
  hist(my_data)
})
```

While the code above looks to be OK from R-programming point of view, it results in an error message. Now reactive value is called outside of reactive function, this is not allowed. Shortly, you will see a way to fix the above mistake.

Reactive functions

There are six types of reactive functions in the reactive toolkit. All of them use a code to build (and rebuild) an object in response to a change in reactive value or multiple reactive values.

`render*()`

First type or group of reactive functions is `render*()` functions. This group is simple, all its functions create some output on the web page depending on the reactive value and if that value changes they rebuild the output.

```
# make a histogram
renderPlot({ hist(rnorm(input$num)) })

# print summary statistics in verbatim
renderPrint({ summary(rnorm(input$num)) })

# print text
renderText({ paste("Mean of the data is ", mean(rnorm(input$num)), sep = "")   })
```

`reactive()`

Second function is `reactive()`. It creates reactive expression. Consider an example below

```r
library(shiny)

ui <- fluidPage( # define a fluid design page
  sliderInput(inputId = "num", label = "Choose a number", value = 25, min = 1, max = 1000),
  plotOutput(outputId = "my_histogram"),    # placement for histogram, beware of comma
  verbatimTextOutput("my_verbatim")         # placement for verbatim text
)

server <- function(input, output) {
  # generate data and make it a reactive expression
  my_data <- reactive({   rnorm(input$num)  })
  output$my_histogram <- renderPlot({ hist(my_data(), main = "Histogram") })  # plot
  output$my_verbatim <- renderPrint({ summary(my_data()) }) # print summary
}
# you don't need to store the result
appToPlot <- shinyApp(ui = ui, server = server)
```

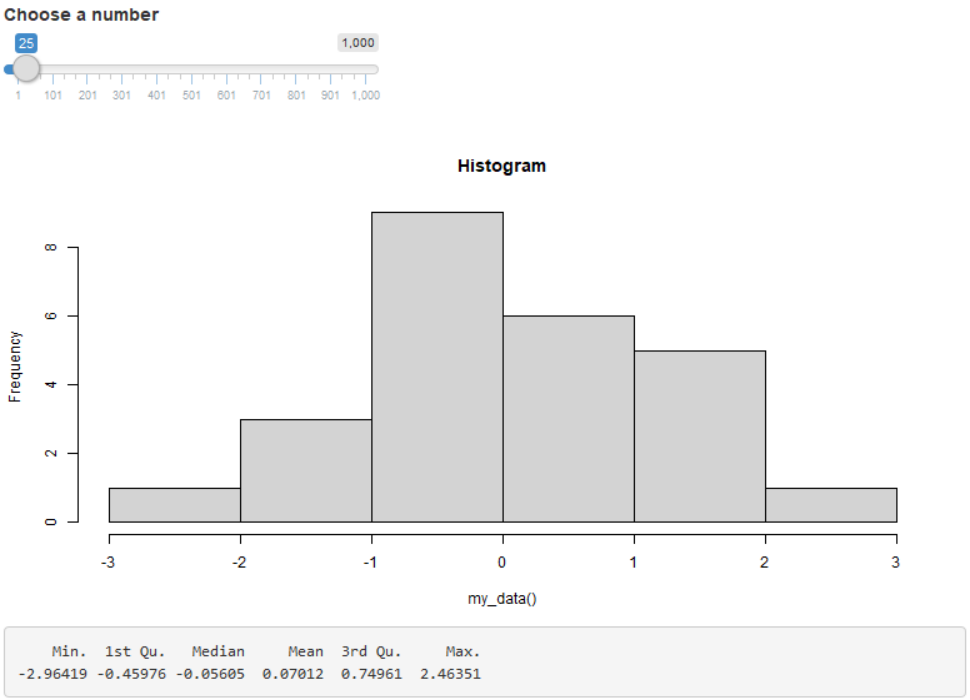


Figure 6: Histogram of normal distribution dashboard

Variable `my_data` is a reactive expression, which acts the same way as a reactive value – that is, it is used inside a reactive function, it invalidates the reactive function and forces the reactive function to update.

The most important difference for us between a reactive expression and a reactive value is that a reactive expression should be called like a function – with round brackets – `my_data()`.

`isolate()`

Next function from the reactive toolkit is a function `isolate()`. It converts a reactive value in to a non-reactive value. This is opposite to what `reactive()` function does converting a non-reactive value in to a

reactive value. If we encapsulate input (that is, reactive) value in `isolate()` function, the reactive function will not respond when you change this input value. Below is the same example as before but with two minor changes

```
library(shiny)

ui <- fluidPage( # define a fluid design page
  sliderInput(inputId = "num", label = "Choose a number", value = 25, min = 1, max = 1000),
  plotOutput(outputId = "my_histogram"),          # placement for histogram, beware of comma
  actionButton(inputId = "update", "Update View", icon("refresh")),  # button to refresh
  verbatimTextOutput("my_verbatim")               # placement for verbatim text
)

server <- function(input, output) {
  # generate data and make it a reactive expression
  my_data <- reactive({   rnorm(input$num)  })
  output$my_histogram <- renderPlot({ hist(my_data()) })    # plot the data
  output$my_verbatim <- renderPrint({
    input$update                      # reactive value
    isolate({ summary(my_data()) })  # isolated reactive value
  })
}
shinyApp(ui = ui, server = server)
```

The button with ID "update" is added to the user interface and the reactive function `renderPrint()` for text output has two reactive values/expressions: `input$update` (from the button) and `my_data()` (generated as before). However, the last one – the reactive expression `my_data()` – is isolated.

It means that the change in the input value `input$num` invalidates function `reactive()` and creates/changes a reactive expression `my_data()`, which in turn invalidates and forces to update function `renderPlot()` but not the function `renderPrint()`.

We can move the slider and each time we get a new histogram, but summary statistics remains the same. Only when we press the button "Update View", the reactive value `input$update` changes and that invalidates `renderPrint()` function and forces it to update. As a result, we get a statistical summary for the latest version of `my_data()` stored in the memory.

`observeEvent()`

The example above introduced an action button. Each time you click on that button, an associated reactive value changes, so it can be used as a trigger to update `render*()` function. Alternatively, it can be used to run something on the Shiny server (or in Rstudio that acts as your server) as it creates an *event*. You can run on the server something very simple as `print()` command in the example below or something very complex. All that is possible by using another function form reactive toolkit – `observeEvent()`.

```
library(shiny)

# create shiny app with one button only
ui <- fluidPage(actionButton(inputId = "clicks", label = "Click me"))

server <- function(input, output) {
  observeEvent(input$clicks, {
    print(as.numeric(input$clicks))  # print on the server but not in app
  })
```

```
}

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

App above runs print function on the server and you get some results, but it does not do or change anything in the app output. Think about downloading data or web scrapping – click the button and get the data.

Another similar function is `observe()`. Its syntax is very similar to `render*()` and `reactive()` functions.

```
library(shiny)

ui <- fluidPage( actionButton(inputId = "clicks", label = "Click me"))

server <- function(input, output) {
  observe({               # observe for the change
    input$clicks          # in this reactive variable
    print("get out!")     # and then run this code on the server
  })
}

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

`eventReactive()`

Function `eventReactive()` allows to delay the response of the dashboard to the change of the reactive variable.

```
library(shiny)

ui <- fluidPage( # define a fluid design page
  sliderInput(inputId = "num", label = "Choose a number", value = 25, min = 1, max = 1000),
  plotOutput(outputId = "my_histogram"),          # placement for histogram, beware of comma
  actionButton(inputId = "update", "Update View", icon("refresh")),  # button to refresh
  verbatimTextOutput("my_verbatim")               # placement for verbatim text
)

server <- function(input, output) {
  # generate data and make it a reactive expression
  # but only if there is an event "the button is clicked"
  my_data <- eventReactive(input$update, { rnorm(input$num) }, ignoreNULL = FALSE)
  output$my_histogram <- renderPlot({ hist(my_data()) })     # plot the data
  output$my_verbatim <- renderPrint({ summary(my_data()) })  # print summary
}
shinyApp(ui = ui, server = server)
```

Input value `input$num` behaves as it is isolated, as a result, the reactive expression `my_data()` is not created when we change `input$num`. The app will wait till we click the button and change reactive value `input$update`. Then reactive expression `my_data()` is created using the latest value of `input$num` stored in the memory.

As you have seen above, the value `input$update` of the action button is an increasing counts of clicks. But the very first value of `input$update` before we make a click is `NULL`. Parameter `ignoreNULL = FALSE` in the function `eventReactive()` tells to run the code even for `NULL` value. That ensures to have a histogram

and statistical summary on the start of the shiny app. If parameter `ignoreNULL = TRUE`, which is a default value for the function argument, then there will be no histogram or summary on the start of the app.

`reactiveValue()`

Shiny does not allow us to change input values in the code. Reminder: input values are reactive values that can activate reactive functions and force them to update. However, we can create our own reactive values to use inside reactive functions and create nice outputs. Function `reactiveValue()` allows to define a reactive value of format `rv$valueID`. Name `rv` works the same way as `input` and it can be absolutely anything. We can change our own custom reactive value and all reactive functions that use this value will be forced to update.

```
library(shiny)

ui <- fluidPage(
  # create drop down menu
  selectInput(inputId = "distr", label = "Distribution:",
              choices = c("Normal" = "norm", "Uniform" = "unif")),
  plotOutput("hist") # histogram placement
)

server <- function(input, output) {

  # create reactive value with ID "data" and a default value
  rv <- reactiveValues(data = 0)

  # change the actual value inside custom reactive value variable
  observeEvent(input$distr, {
    if(input$distr == "norm"){
      rv$data <- rnorm(1000)
    } else if(input$distr == "unif"){
      rv$data <- runif(1000)
    }
  })

  # plot the histogram if reactive value changes
  output$hist <- renderPlot({
    hist(rv$data)
  })
}

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

Here the input reactive value is `input$distr`. Its change invalidates `observeEvent()` function and force it to change variable `rv$data`. And as `rv$data` is a reactive value, its change in turn invalidates function `renderPlot()` and force it to respond – update the graph.

Customise User Interface

Besides a number of different ways how shiny app can behave, there is a number of different ways how it can look like. Again, if you need an inspiration on the shiny app design, have a look on the gallery of dashboards created using `shiny` – https://shiny.rstudio.com/gallery/ . Most examples are well documented, so you can see how they work and use the same ideas for your own dashboard.

Section of user interface creates an HTML/CSS code. There are many functions to create different type of design for shiny app (check Function reference, section UI Layout) and all of them result in HTML/CSS code. You can try to run these function alone and see the result.

```
library(shiny)

# create user interface
ui <- fluidPage("Test Page")

# have a look on the HTML code
print(ui)
```

```
## <div class="container-fluid">Test Page</div>
```

This is not so exciting web page. You can try any of the previous examples

```
library(shiny)

# create user interface with drop down menu and a plot
ui <- fluidPage(
  # create drop down menu
  selectInput(inputId = "distr", label = "Distribution:",
              choices = c("Normal" = "norm", "Uniform" = "unif")),
  plotOutput("hist") # histogram placement
)

# have a look on the HTML code
print(ui)
```

```
## <div class="container-fluid">
##   <div class="form-group shiny-input-container">
##     <label class="control-label" id="distr-label" for="distr">Distribution:</label>
##     <div>
##       <select id="distr"><option value="norm" selected>Normal</option>
## <option value="unif">Uniform</option></select>
##       <script type="application/json" data-for="distr" data-nonempty="">{"plugins":["selectize-plugi
##     </div>
##   </div>
##   <div id="hist" class="shiny-plot-output" style="width:100%;height:400px;"></div>
## </div>
```

This code is more complex and it might be extremely complex if you create a complex dashboard with a large number of different elements/objects. At the same time, you don't need to know too much of HTML/CSS (while this would be handy) as `shiny` functions will create a code for you.

Below are several different ways how you can customise an appearance of the dashboard.

Static dashboard content - HTML/CSS code

You already know how to add some reactive objects to the dashboard – input and output objects. However, any dashboard needs some static content too, e.g. headings, comments, links. There is an object `tags` in the package `shiny`. This is a list of HTML functions. It is a real data type "list" common for R and its elements are functions. The syntax is following:

```
# call for the function out of the tags list and provide text to include
# in that tag as function argument
print(tags$b("Test Page"))
```

```
## <b>Test Page</b>
```

We use dollar-notation to get an element from the list by the name of this element; and round brackets with parameters at the end as each element of the list is a function.

The most common and the most popular HTML tags have their own functions and can be used directly. For example headings of levels 1 and 2

```
# call for the function out of the tags list
print(tags$h1("Heading level 1"))
```

```
## <h1>Heading level 1</h1>
```

```
# use a function
print(h2("Heading level 2"))
```

```
## <h2>Heading level 2</h2>
```

Either way you have to use round brackets as `tags$h1()` and `h1()` are functions.

To see the list of all HTML-creating functions available from the list `tags` you can use function `names()` as with any other list. This is a very long list.

```
# get list of HTML functions
names(tags)
```

```
##   [1] "a"                "abbr"             "address"
##   [4] "animate"          "animateMotion"    "animateTransform"
##   [7] "area"             "article"          "aside"
##  [10] "audio"            "b"                "base"
##  [13] "bdi"              "bdo"              "blockquote"
##  [16] "body"             "br"               "button"
##  [19] "canvas"           "caption"          "circle"
##  [22] "cite"             "clipPath"         "code"
##  [25] "col"              "colgroup"         "color-profile"
##  [28] "command"          "data"             "datalist"
##  [31] "dd"               "defs"             "del"
##  [34] "desc"             "details"          "dfn"
##  [37] "dialog"           "discard"          "div"
##  [40] "dl"               "dt"               "ellipse"
##  [43] "em"               "embed"            "eventsource"
```

```
##  [46] "feBlend"             "feColorMatrix"       "feComponentTransfer"
##  [49] "feComposite"         "feConvolveMatrix"    "feDiffuseLighting"
##  [52] "feDisplacementMap"   "feDistantLight"      "feDropShadow"
##  [55] "feFlood"             "feFuncA"             "feFuncB"
##  [58] "feFuncG"             "feFuncR"             "feGaussianBlur"
##  [61] "feImage"             "feMerge"             "feMergeNode"
##  [64] "feMorphology"        "feOffset"            "fePointLight"
##  [67] "feSpecularLighting"  "feSpotLight"         "feTile"
##  [70] "feTurbulence"        "fieldset"            "figcaption"
##  [73] "figure"              "filter"              "footer"
##  [76] "foreignObject"       "form"                "g"
##  [79] "h1"                  "h2"                  "h3"
##  [82] "h4"                  "h5"                  "h6"
##  [85] "hatch"               "hatchpath"           "head"
##  [88] "header"              "hgroup"              "hr"
##  [91] "html"                "i"                   "iframe"
##  [94] "image"               "img"                 "input"
##  [97] "ins"                 "kbd"                 "keygen"
## [100] "label"               "legend"              "li"
## [103] "line"                "linearGradient"      "link"
## [106] "main"                "map"                 "mark"
## [109] "marker"              "mask"                "menu"
## [112] "meta"                "metadata"            "meter"
## [115] "mpath"               "nav"                 "noscript"
## [118] "object"              "ol"                  "optgroup"
## [121] "option"              "output"              "p"
## [124] "param"               "path"                "pattern"
## [127] "picture"             "polygon"             "polyline"
## [130] "pre"                 "progress"            "q"
## [133] "radialGradient"      "rb"                  "rect"
## [136] "rp"                  "rt"                  "rtc"
## [139] "ruby"                "s"                   "samp"
## [142] "script"              "section"             "select"
## [145] "set"                 "slot"                "small"
## [148] "solidcolor"          "source"              "span"
## [151] "stop"                "strong"              "style"
## [154] "sub"                 "summary"             "sup"
## [157] "svg"                 "switch"              "symbol"
## [160] "table"               "tbody"               "td"
## [163] "template"            "text"                "textarea"
## [166] "textPath"            "tfoot"               "th"
## [169] "thead"               "time"                "title"
## [172] "tr"                  "track"               "tspan"
## [175] "u"                   "ul"                  "use"
## [178] "var"                 "video"               "view"
## [181] "wbr"
```

A collection of common HTML functions that do not need a wrapper of list `tags` is much shorter:

- `h1()` to `h6()` for six levels of headings
- `p()` for a paragraph
- `br()` for a line break
- `strong()` for a bold text
- `em()` for italicised text

- `a()` for a hyperlink
- `hr()` for a horizontal line
- `code()` for a verbatim text
- `img()` to include an image

The last function `img()` requires a special treatment which is not common for R functions. It has to have its arguments named and if you use local images stored on your computer (as an opposite to using hyperlinks to images on the Internet), the function will look for these images in folder `www` in the same directory as your shiny app. You don't need to include `www` in the code.

```
# file test.png should be in folder www
print(img(height = 100, width = 100, src = "test.png"))
```

```
## <img height="100" width="100" src="test.png"/>
```

If you really want, you can write HTML code manually and include it in the shiny app. You store your HTML code as a character and use function `HTML()`

```
# manually created HTML code
my_code <- "<h1>Test Page</h1>
<p>This is a test page for manual html code.</p>"

# shiny app user interface
ui <- fluidPage(
  HTML(my_code)
)

# check the result
print(ui)
```

```
## <div class="container-fluid"><h1>Test Page</h1>
## <p>This is a test page for manual html code.</p></div>
```

Dashboard layout

Besides visual elements like headings, images, text, you might want to have a better control of the overall layout of the dashboard. Place input slider on the top or bottom or left or right, above or below the graph, on the left or right side of the graph, etc.

Page design

There are two main type of pages – `fluidPage()` and `fixedPage()`. The difference is clear from the names: we can get a flexible layout that tries to fill all available space or fixed size layout. As you never know what screen size or browser size are on the user's computer, then `fluidPage()` is way more popular option.

There are several specialised page layouts with higher-level functionality, e.g. `sidebarLayout` (reviewed later) that can be used only with fluid page design. If you want to get the same visual design (menus or tabs) for the fixed page, you need to create it manually.

Grid design with `fluidRow()` / `fixedRow()` and `column()`

While the page layout is (typically) flexible, it might have a grid structure. Horizontal sections of that grid are created by function `fluidRow()` (or `fixedRow()` for fixed size page). Vertical sections are created by function `column()` with width and offset arguments.

```
library(shiny)

ui <- fluidPage( # define a fluid design page
  # top level heading, beware the commas
  h1("Grid structure"),
  # first horizontal section
  fluidRow(
    # column of width 3, remember the commas between columns
    column(3, wellPanel(p("Width 3"))),
    # next column of width 6
    column(6, wellPanel(p("Width 6")))),
  # second horizontal section without any columns
  fluidRow(
    wellPanel(p("No columns"))
  ),
  # third horizontal section with one column offset to the right width 4
  fluidRow(
    column(4, wellPanel(p("Width 4 offset 6")), offset = 6)
  )
)

server <- function(input, output) {
  # nothing here as this example for layout only
}
# run the app
appToPlot <- shinyApp(ui = ui, server = server)
```

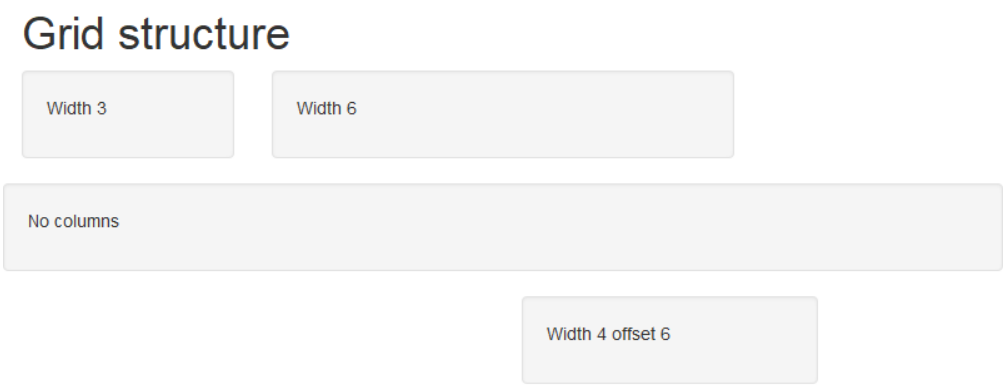


Figure 7: Shiny app grid structure

Total width of columns is 12. More precisely, every row can have up to 12 column width. You can have two columns of width 6; or three columns of width 4; and so on. You can have a column with an offset, hence there will be nothing on the left of the column.

Grid structure can be split further than 12 units if you want. Every column can have its own grid structure with `fluidRow()` and `column()`. Each "smaller" row can be split in 12 units too.

An element created by function `wellPanel()` in the examples above and below is used to illustrate boundaries of the rows and columns only. The grid layout works the same way even if you don't see the grid itself. However, if the window for the app is too narrow to properly fit the grid layout, then the entire layout of the app will be fitted in one column.

```
library(shiny)

ui <- fluidPage( # define a fluid design page
  h1("Grid structure"),
  # first row with two equal size columns
  fluidRow(
    column(6, wellPanel(p("Width 6"))),
    column(6, wellPanel(p("Width 6")))
  ),
  # second row with two equal size columns again
  fluidRow(
    column(6, wellPanel(p("Width 6"))),
    column(6,
           # row section inside second column with two column of size 3 and 9
           fluidRow(
             column(3, wellPanel(p("Width 3"))),
             column(9, wellPanel(p("Width 9")))
    ) )
  )
)

server <- function(input, output) {
  # nothing here as this example for layout only
}
# run the app
appToPlot <- shinyApp(ui = ui, server = server)
```

Figure 8: Shiny app grid structure

An alternative approach to organise rows and columns is to use functions `fillRow()` and `fillCol()`. They allow multiple elements placed in each row and/or columns and hight and width of rows and columns defined as proportion of the parent object size (in percentage). However, these functions can be used only inside `fillPage()` page layout.

Panels and tabs

To make some elements of the design more prominent than others, e.g. to drag attention to the input area of the dashboard, you can use panels. For example, the dashboard created before can look a bit better organised with 'wellPanel()' – a panel with a nice grey background.

```
library(shiny)

ui <- fluidPage(
  wellPanel( # put input object into wellPanel() function
  sliderInput(inputId = "num", label = "Choose a number", value = 25, min = 1, max = 1000)),
  plotOutput(outputId = "my_histogram"),    # placement for histogram, beware of comma
  verbatimTextOutput("my_verbatim")         # placement for verbatim text
)

server <- function(input, output) {
  my_data <- reactive({   rnorm(input$num)  })
  output$my_histogram <- renderPlot({ hist(my_data(), main = "Histogram") })  # plot
  output$my_verbatim <- renderPrint({ summary(my_data()) }) # print summary
}
# you don't need to store the result
appToPlot <- shinyApp(ui = ui, server = server)
```

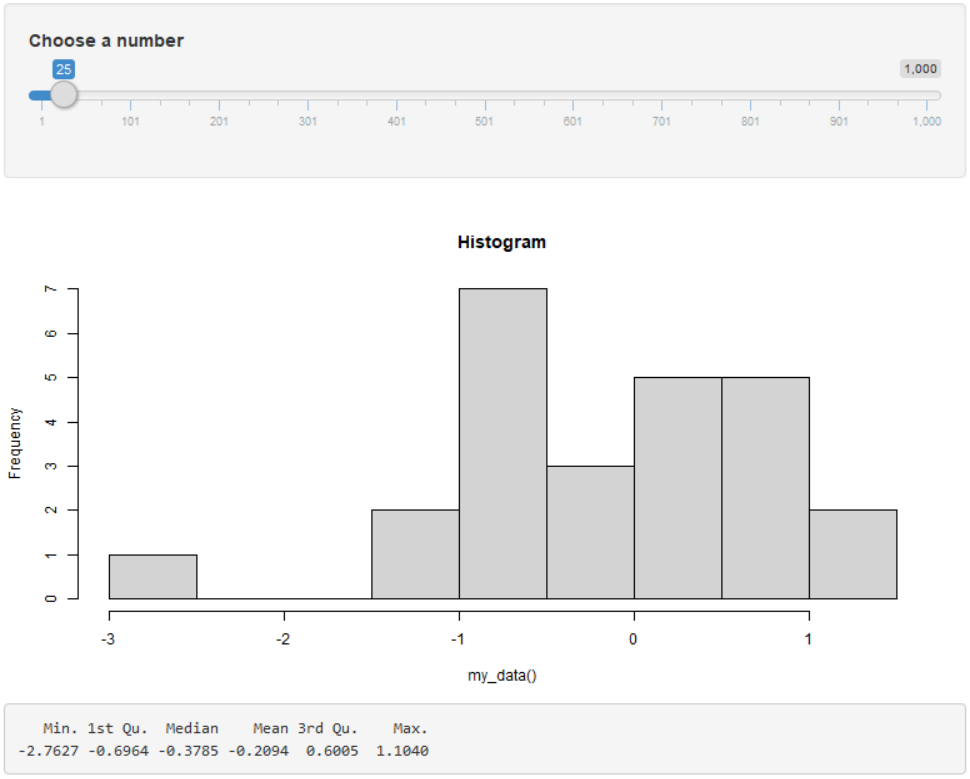


Figure 9: Histogram of normal distribution dashboard

There are a lot of different panels. Some of them can be used independently as `wellPanel()`, others can be used only in a combination with other functions to create some particular design, like side menu layout below.

```
library(shiny)

ui <- fluidPage(
  # dashboard title
  titlePanel("Histogram of Normal Distribution"),

  # page layout design
  sidebarLayout(
    # sidebar with a slider input
    sidebarPanel(sliderInput(inputId = "num", label = "Choose a number",
                             value = 25, min = 1, max = 1000) ),
    # main window with outputs
    mainPanel(
      plotOutput(outputId = "my_histogram"),
      verbatimTextOutput(outputId = "my_verbatim")
  ) ) )

# no changes to server function
server <- function(input, output) {
  my_data <- reactive({   rnorm(input$num)  })
  output$my_histogram <- renderPlot({ hist(my_data(), main = "Histogram") })  # plot
  output$my_verbatim <- renderPrint({ summary(my_data()) }) # print summary
}
# you don't need to store the result
appToPlot <- shinyApp(ui = ui, server = server)
```

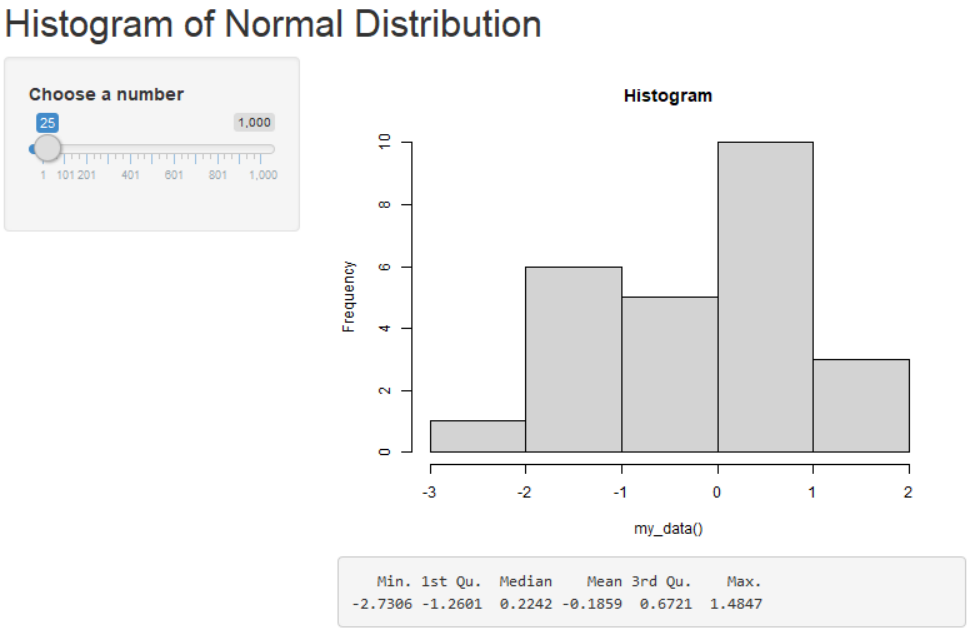


Figure 10: Histogram of normal distribution dashboard

Next type of panels is `tabPanel()`. It allows to have a stack of many panels. Each tab-panel has its own user interface and each panel is available as a tab. Tabs can be organised together by functions `tabsetPanel()`, `navlistPanel()`, `navbarPanel()`.

```
library(shiny)
ui <- fluidPage(
  # layout of tabs
  tabsetPanel(
    # three separate tabs / pages
    tabPanel("Normal", plotOutput(outputId = "norm")),
    tabPanel("Uniform", plotOutput(outputId = "unif")),
    tabPanel("Exponential", plotOutput(outputId = "exp"))
  )
)
server <- function(input, output) {
  output$norm <- renderPlot({ hist(rnorm(200)) })
  output$unif <- renderPlot({ hist(runif(200)) })
  output$exp <- renderPlot({ hist(rexp(200)) })
}
# you don't need to store the result
appToPlot <- shinyApp(ui = ui, server = server)
```

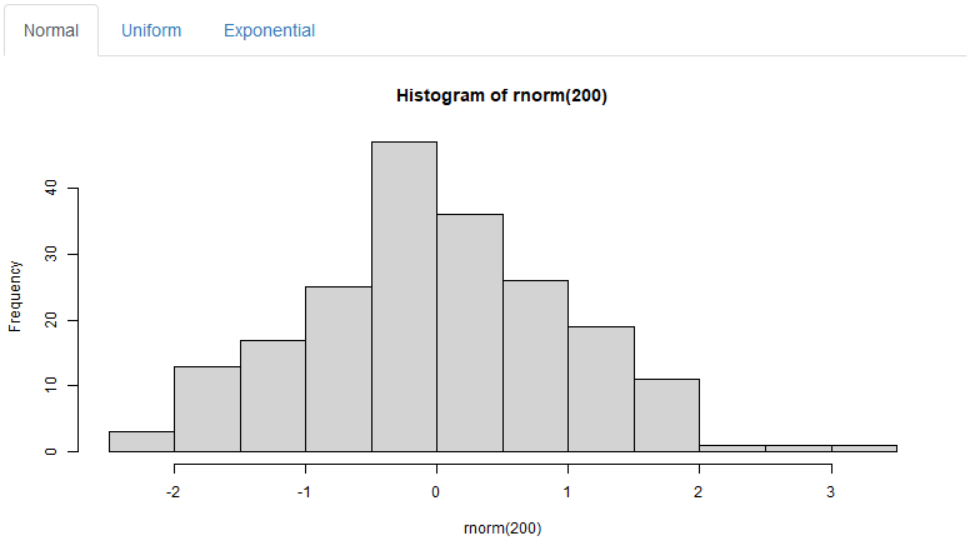


Figure 11: Distributions dashboard

The same code as above with one minor change: `navlistPanel()` instead of `tabsetPanel()`

```
library(shiny)
ui <- fluidPage(
  # navigation bar for tab-panels
  navlistPanel(
    # three separate tabs / pages
    tabPanel("Normal", plotOutput(outputId = "norm")),
    tabPanel("Uniform", plotOutput(outputId = "unif")),
    tabPanel("Exponential", plotOutput(outputId = "exp")),
    widths = c(2,10) )  # widths for menu and for output section
)
server <- function(input, output) {
  output$norm <- renderPlot({ hist(rnorm(200)) })
```

```
  output$unif <- renderPlot({ hist(runif(200)) })
  output$exp <- renderPlot({ hist(rexp(200)) })
}
# you don't need to store the result
appToPlot <- shinyApp(ui = ui, server = server)
```

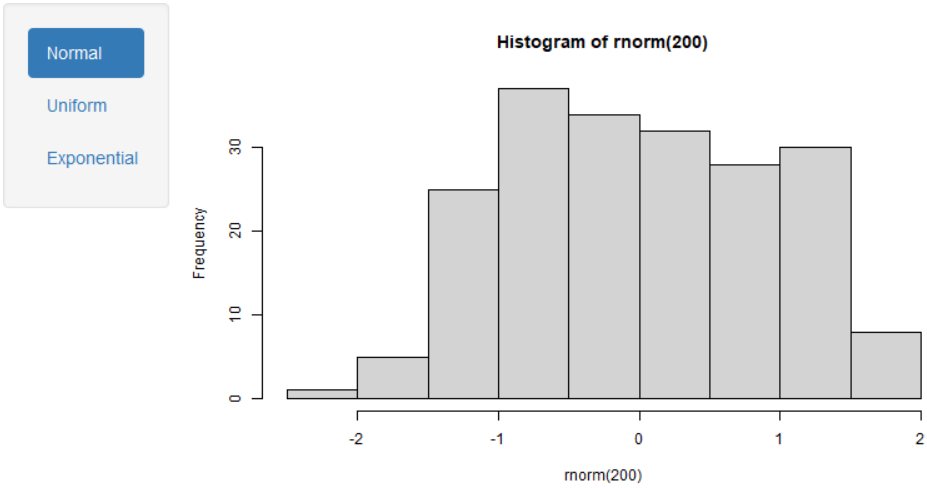


Figure 12: Distributions dashboard

As a result, you get a navigation menu on the side.

Prepackaged layouts and supporting packages

An alternative approach to the creating your own designs is to use some of repackaged designs. For example, tab-based design:

```
library(shiny)
# new page function to replace fluidPage
ui <- navbarPage( title = "Distributions",
    # three separate tabs / pages
    tabPanel("Normal", plotOutput(outputId = "norm")),
    tabPanel("Uniform", plotOutput(outputId = "unif")),
    tabPanel("Exponential", plotOutput(outputId = "exp"))
)
server <- function(input, output) {
  output$norm <- renderPlot({ hist(rnorm(200)) })
  output$unif <- renderPlot({ hist(runif(200)) })
  output$exp <- renderPlot({ hist(rexp(200)) })
}
# you don't need to store the result
appToPlot <- shinyApp(ui = ui, server = server)
```

There are prepared layouts in the other packages that extends functionality of `shiny` package. For example, package `shinydashboard` allows to create a good looking dashboard with minimal coding.

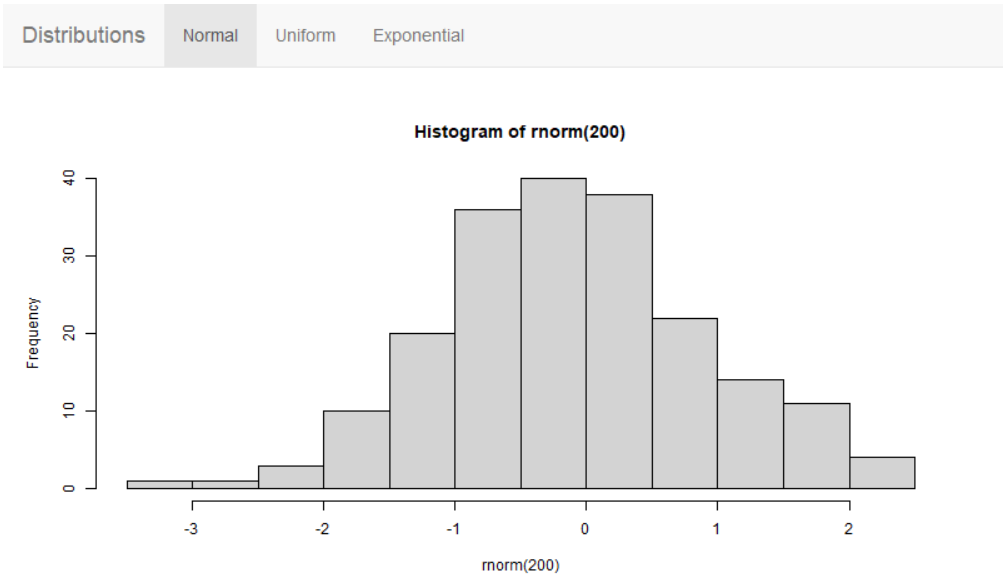


Figure 13: Distributions dashboard

```
library(shiny)
library(shinydashboard)
```

```
## Warning: package 'shinydashboard' was built under R version 4.0.5
```

```
##
## Attaching package: 'shinydashboard'
```

```
## The following object is masked from 'package:graphics':
##
##     box
```

```
# new type of the page
ui <- dashboardPage(
  # header
  dashboardHeader(title = "Distributions"),
  # side menu
  dashboardSidebar( sidebarMenu(
    menuItem("Normal", tabName = "norm"),
    menuItem("Uniform", tabName = "unif"),
    menuItem("Exponential", tabName = "exp"))),
  # dashboard body with three stacked panels
  dashboardBody( tabItems(
    # three separate tabs / pages
    tabItem("norm", fluidRow(
      h1("Normal"),
      plotOutput(outputId = "norm"))),
    tabItem("unif", fluidRow(
      h1("Uniform"),
      plotOutput(outputId = "unif"))),
```

```
    tabItem("exp", fluidRow(
      h1("Exponential"),
      plotOutput(outputId = "exp")))
  ))
)
# no changes to server function
server <- function(input, output) {
  output$norm <- renderPlot({ hist(rnorm(200)) })
  output$unif <- renderPlot({ hist(runif(200)) })
  output$exp <- renderPlot({ hist(rexp(200)) })
}
# you don't need to store the result
appToPlot <- shinyApp(ui = ui, server = server)
```

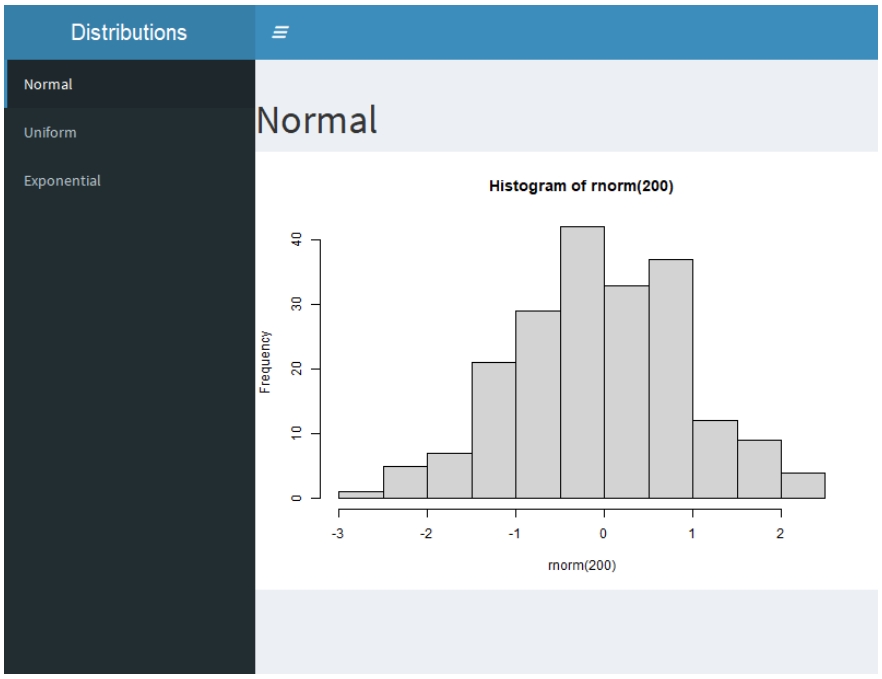


Figure 14: Distributions dashboard

If you add extra functionality of `dashboardthemes` package (https://github.com/nik01010/dashboardthemes) then you can change the above dashboard with one line of code only.

```
# the same code as above
library(shiny)
library(shinydashboard)
library(dashboardthemes) # new package
```

```
## Warning: package 'dashboardthemes' was built under R version 4.0.5
```

```
ui <- dashboardPage(
  dashboardHeader(title = "Distributions"),
  dashboardSidebar( sidebarMenu(
    menuItem("Normal", tabName = "norm"),
```

```
    menuItem("Uniform", tabName = "unif"),
    menuItem("Exponential", tabName = "exp"))),
  dashboardBody(
    shinyDashboardThemes(theme = "flat_red"), # new line of code
    tabItems(
    tabItem("norm", fluidRow(h1("Normal"),  plotOutput(outputId = "norm"))),
    tabItem("unif", fluidRow(h1("Uniform"), plotOutput(outputId = "unif"))),
    tabItem("exp", fluidRow(h1("Exponential"), plotOutput(outputId = "exp")))
  ))
)
# no changes to server function
server <- function(input, output) {
  output$norm <- renderPlot({ hist(rnorm(200)) })
  output$unif <- renderPlot({ hist(runif(200)) })
  output$exp <- renderPlot({ hist(rexp(200)) })
}
# you don't need to store the result
appToPlot <- shinyApp(ui = ui, server = server)
```

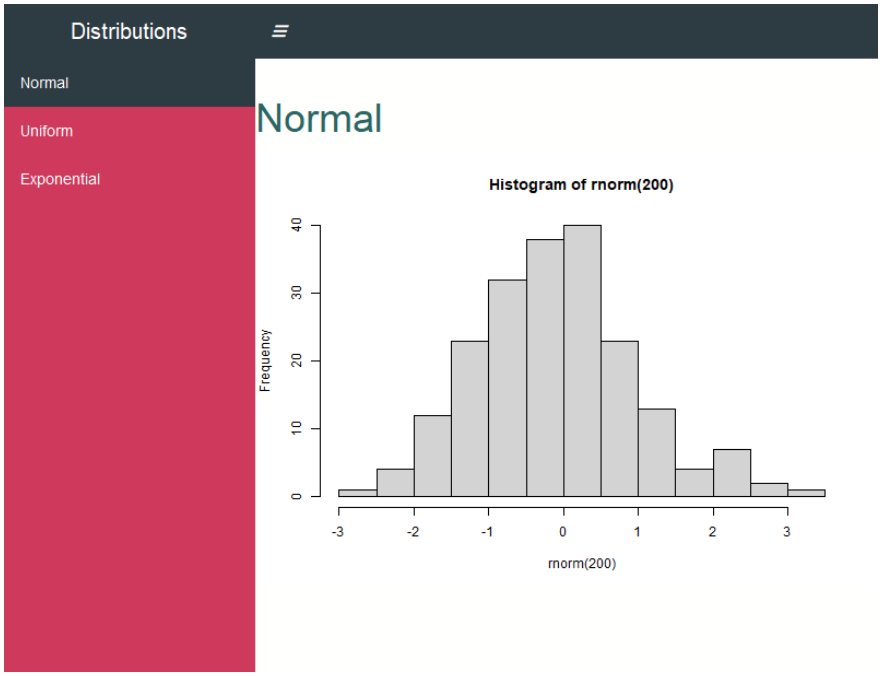


Figure 15: Distributions dashboard

Another package with themes fo shiny dashboards is **shinythemes**. It works with "original" shiny packages. One extra line of code and you change the theme from defualt grey into something more optimistic.

```
library(shiny)
library(shinythemes) # new package
```

```
## Warning: package 'shinythemes' was built under R version 4.0.5
```

```r
ui <- navbarPage( title = "Distributions",
    tabPanel("Normal", plotOutput(outputId = "norm")),
    tabPanel("Uniform", plotOutput(outputId = "unif")),
    tabPanel("Exponential", plotOutput(outputId = "exp")),
    theme = shinytheme("cerulean")   # extra argument
)
server <- function(input, output) {
  output$norm <- renderPlot({ hist(rnorm(200)) })
  output$unif <- renderPlot({ hist(runif(200)) })
  output$exp <- renderPlot({ hist(rexp(200)) })
}
# you don't need to store the result
appToPlot <- shinyApp(ui = ui, server = server)
```

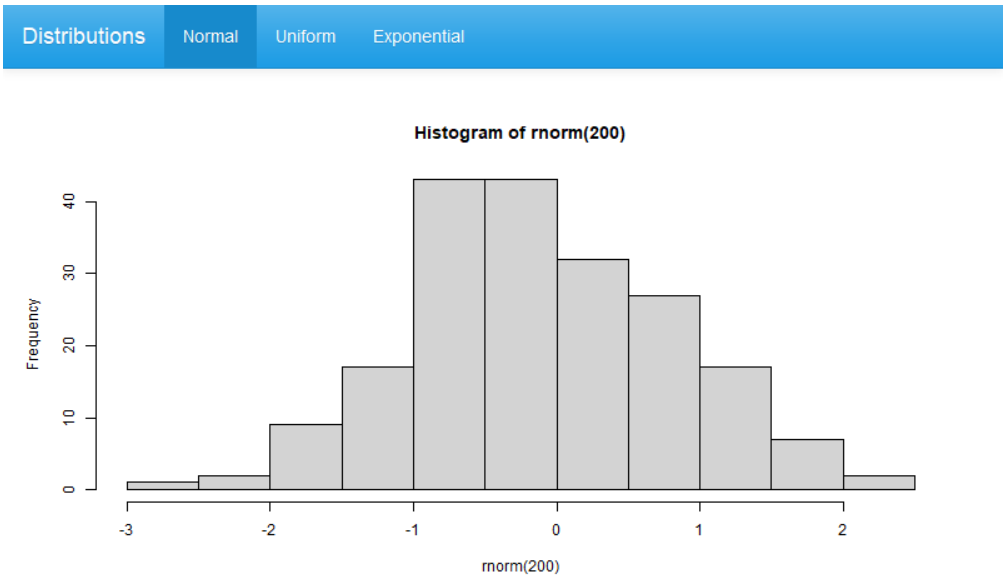


Figure 16: Distributions dashboard

Another collection of dashboard themes with support for `shiny` and `shinydashboard` is package `fresh` (https://github.com/dreamRs/fresh).

Shiny app design is based on CSS, hence by changing styles you can change absolutely anything. You can do it manually but ready-made packages is an easy way.