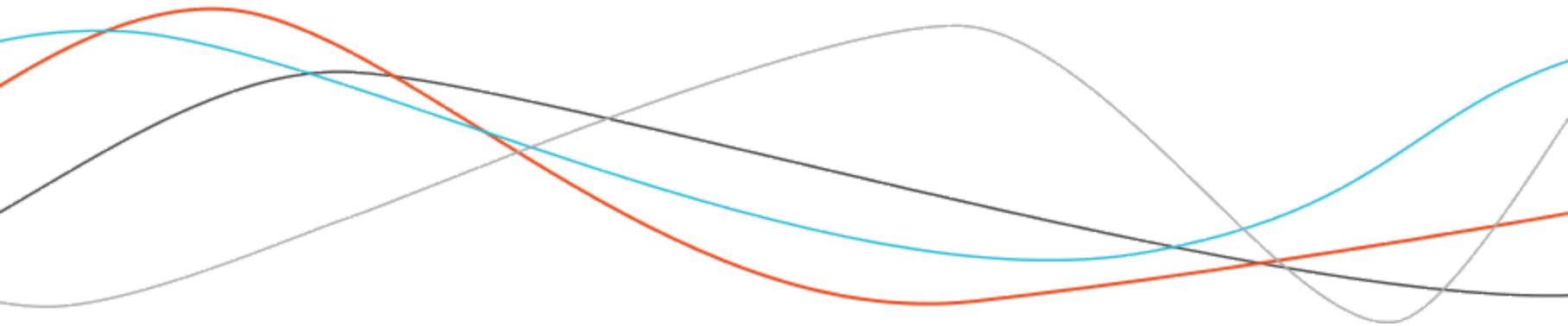
THINKR

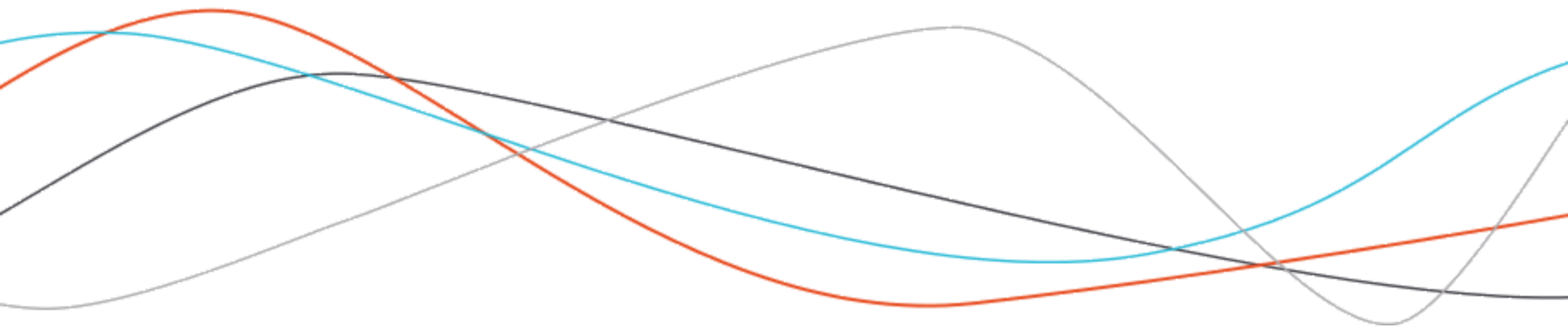
R Shiny

November 19, 2018



Introduction

webapp and dashboards



Shiny: framework of *web applications* for R

The success of the package (for R users) lies in:

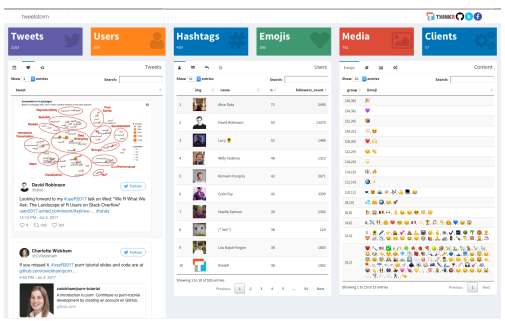
- its ease of writing: no web development skills are required
- the computational and modelling potential of R is easily mobilized: it is easy to start from an R analysis script to create an application
- applications are published locally or on the internets via a shiny server

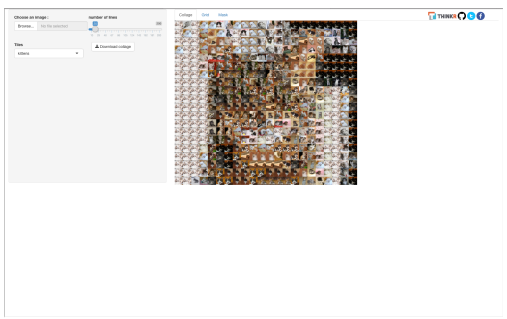
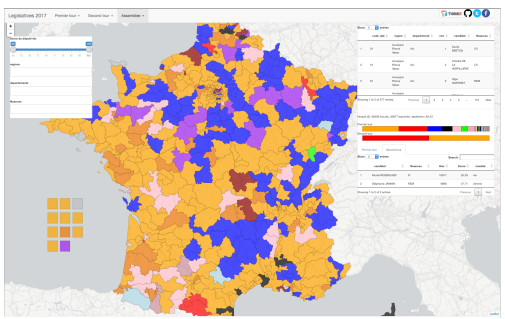
Shiny

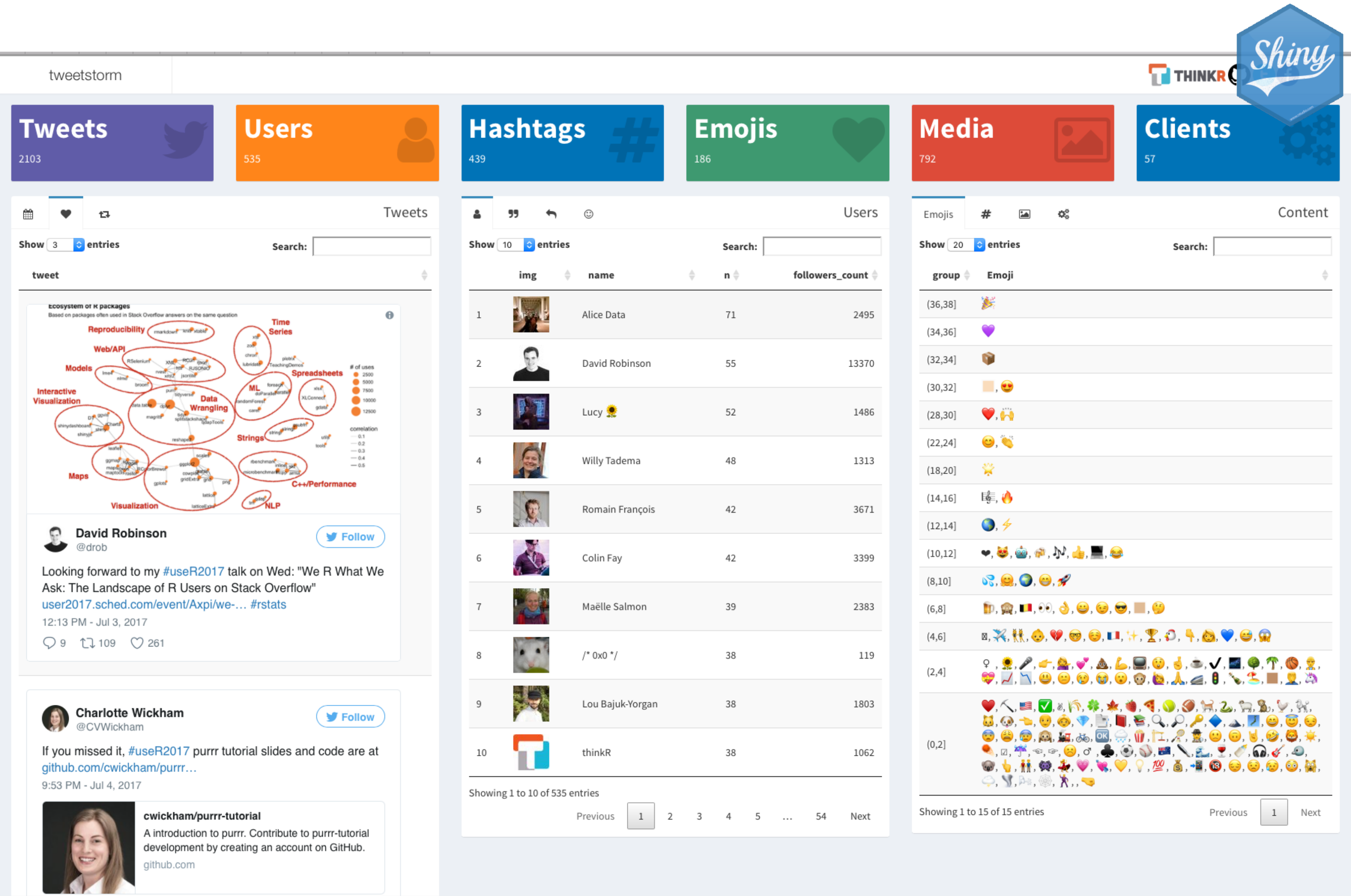
- is a rstudio product
- arrived in the world R in 2012
- is everywhere in conferences and R meetups
- even has its own conference

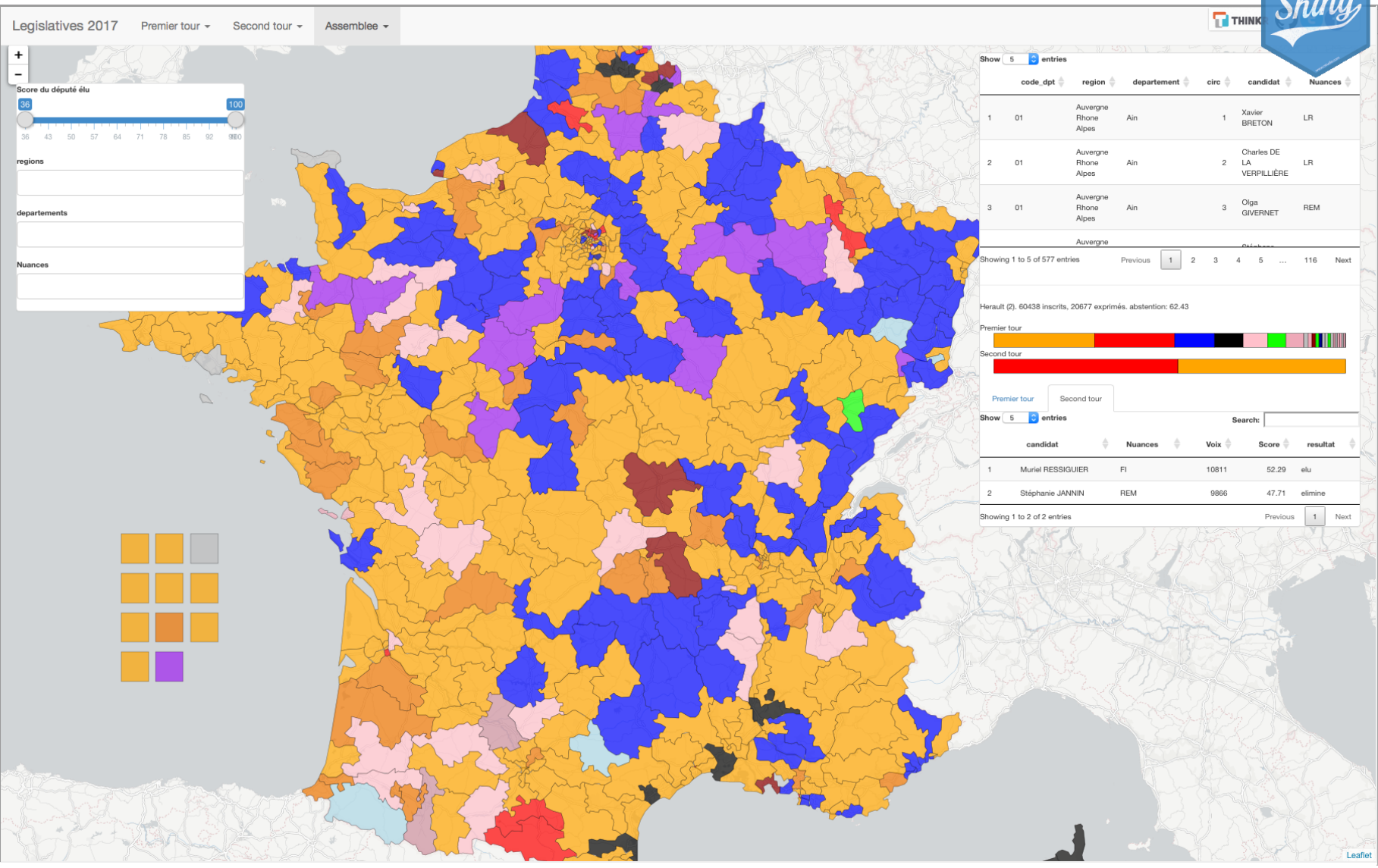
Overview

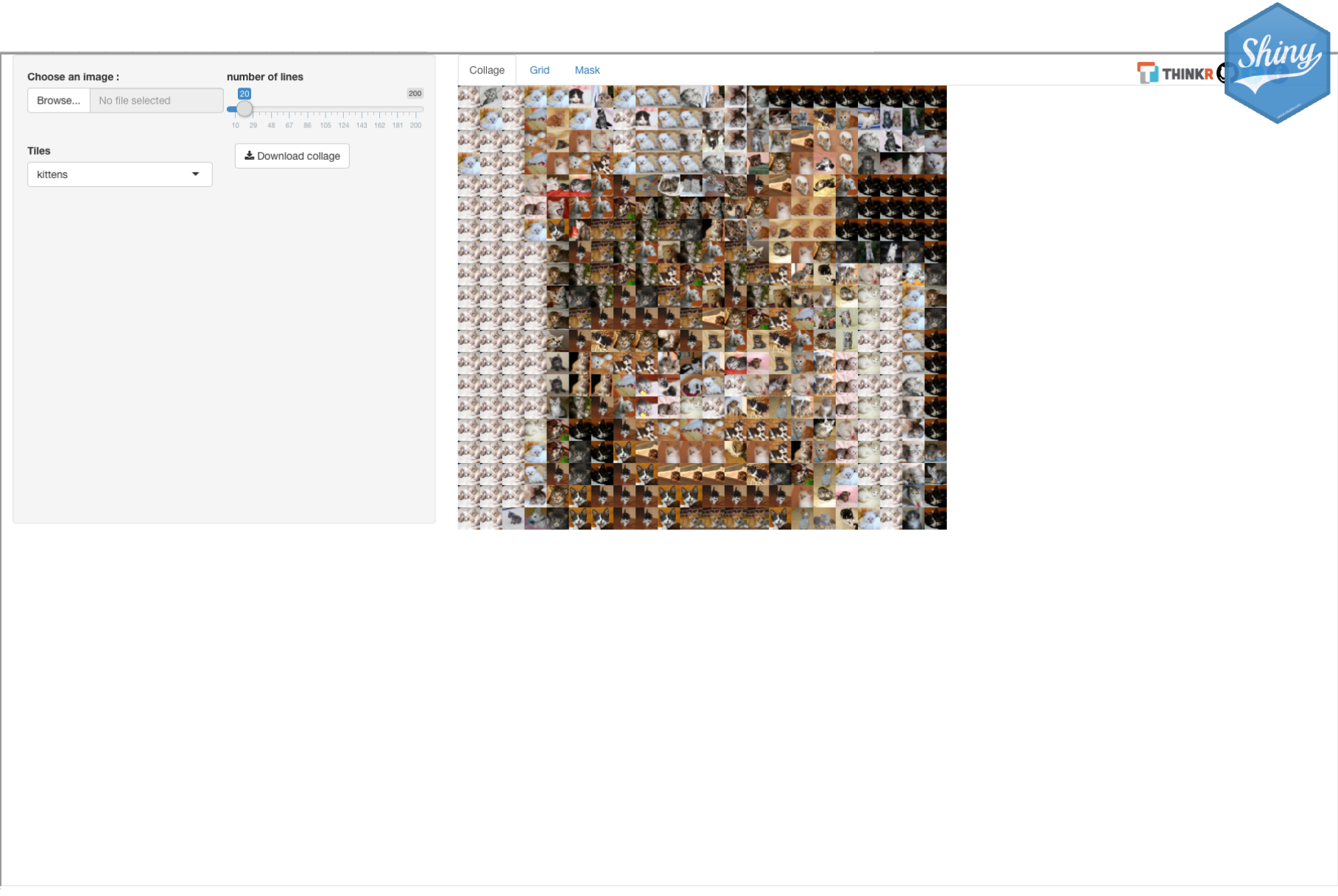
- tweetstorm : dashboard of the activity on twitter during useR2017
- collage : photo collage tool
- legislative2017 : analysis of the results of the 2017 legislative elections
- Kmeans : k nearest neighbors
- Density estimation : density estimation




tweetstorm
THINKR
Shiny
Tweets
2103
Users
535
Hashtags
439
Emojis
186
Media
792
Clients
57
Tweets
Show 3 entries
Search:
tweet
Ecosystem of R packages
Based on packages often used in Stack Overflow answers on the same question
Reproducibility
Time Series
Web/API
Models
Spreadsheets
Interactive Visualization
ML
Data Wrangling
Strings
Maps
C++/Performance
Visualization
NLP
David Robinson
@drob
Follow
Looking forward to my #useR2017 talk on Wed: "We R What We Ask: The Landscape of R Users on Stack Overflow" user2017.sched.com/event/Axpi/we-... #rstats
12:13 PM - Jul 3, 2017
9
109
261
Charlotte Wickham
@CVWickham
Follow
If you missed it, #useR2017 purrr tutorial slides and code are at github.com/cwickham/purrr...
9:53 PM - Jul 4, 2017
cwickham/purrr-tutorial
A introduction to purrr. Contribute to purrr-tutorial development by creating an account on GitHub.
github.com
Users
Show 10 entries
Search:
img name n followers_count
1 Alice Data 71 2495
2 David Robinson 55 13370
3 Lucy 52 1486
4 Willy Tadema 48 1313
5 Romain François 42 3671
6 Colin Fay 42 3399
7 Maëlle Salmon 39 2383
8 /* 0x0 */ 38 119
9 Lou Bajuk-Yorgan 38 1803
10 thinkR 38 1062
Showing 1 to 10 of 535 entries
Previous 1 2 3 4 5 … 54 Next
Emojis
Content
Show 20 entries
Search:
group Emoji
(36,38]
(34,36]
(32,34]
(30,32]
(28,30]
(22,24]
(18,20]
(14,16]
(12,14]
(10,12]
(8,10]
(6,8]
(4,6]
(2,4]
(0,2]
Showing 1 to 15 of 15 entries
Previous 1 Next

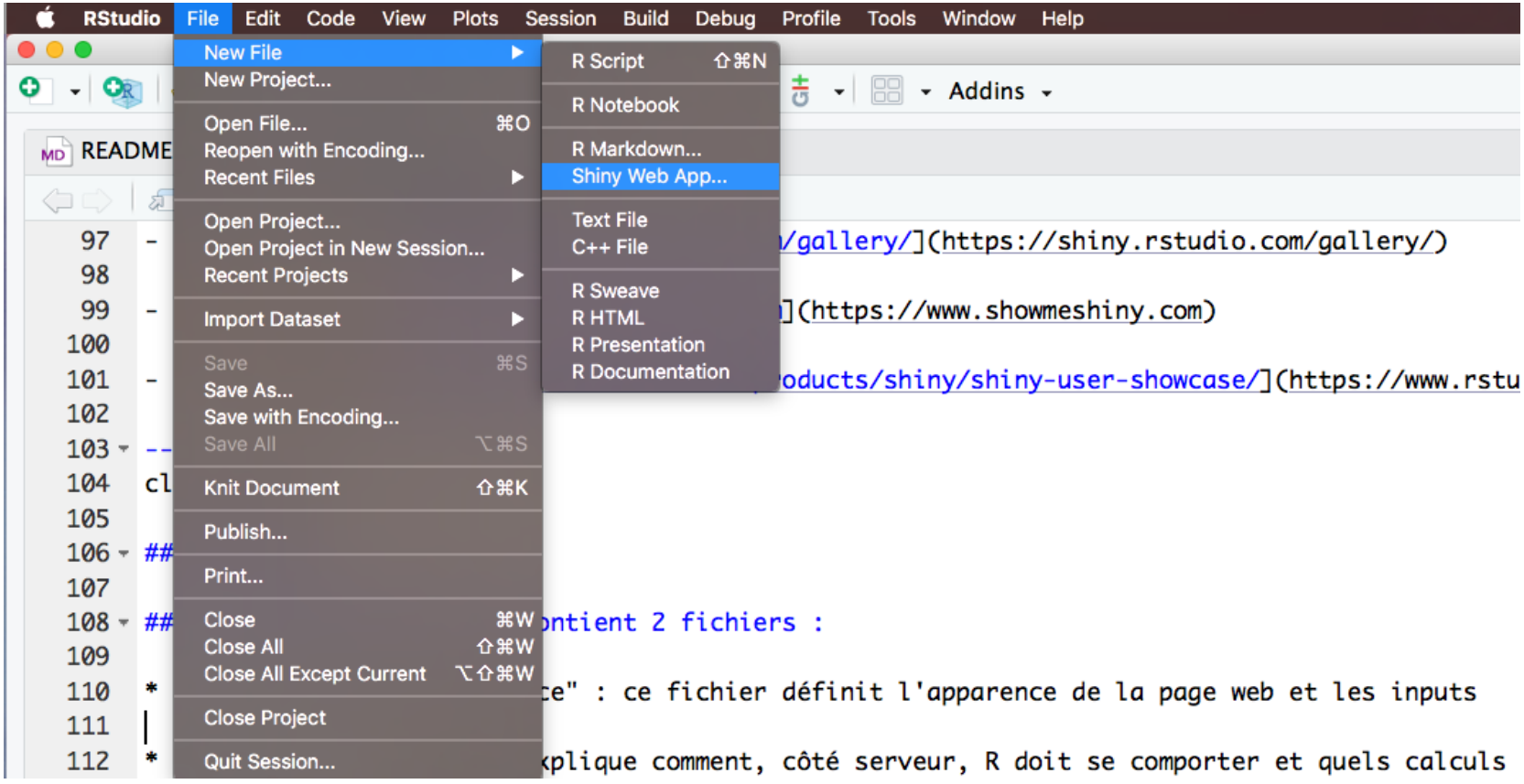

Legislatives 2017
Premier tour
Second tour
Assemblee
Score du député élu
36
100
regions
departements
Nuances
Show 5 entries
code_dpt region departement circ candidat Nuances
1 01 Auvergne Rhone Alpes Ain 1 Xavier BRETON LR
2 01 Auvergne Rhone Alpes Ain 2 Charles DE LA VERPILLIÈRE LR
3 01 Auvergne Rhone Alpes Ain 3 Olga GIVERNET REM
Showing 1 to 5 of 577 entries Previous 1 2 3 4 5 ... 116 Next
Herault (2). 60438 inscrits, 20677 exprimés. abstention: 62.43
Premier tour
Second tour
Premier tour
Second tour
Show 5 entries
Search:
candidat Nuances Voix Score resultat
1 Muriel RESSIGUIER FI 10811 52.29 elu
2 Stéphanie JANNIN REM 9866 47.71 elimine
Showing 1 to 2 of 2 entries Previous 1 Next
Leaflet


Choose an image :
Browse...
No file selected
number of lines
20
200
10 29 48 67 86 105 124 143 162 181 200
Tiles
kittens
Download collage
Collage
Grid
Mask
THINKR
Shiny

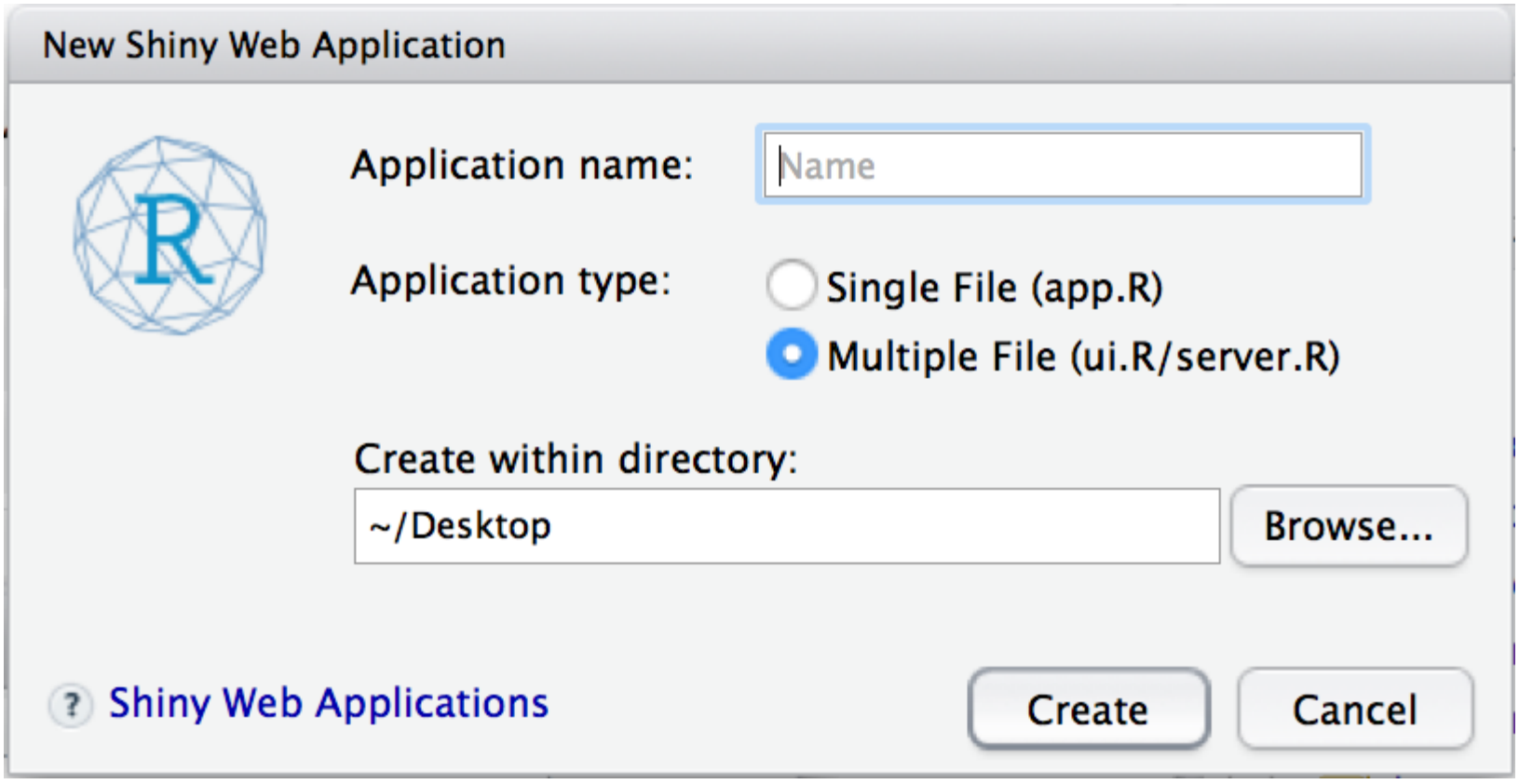
Other examples

- Shiny Gallery: https://shiny.rstudio.com/gallery/
- Show Me Shinyhttps://www.showmeshiny.com
- Shiny showcasehttps://www.rstudio.com/products/shiny/shiny-user-showcase/

first app

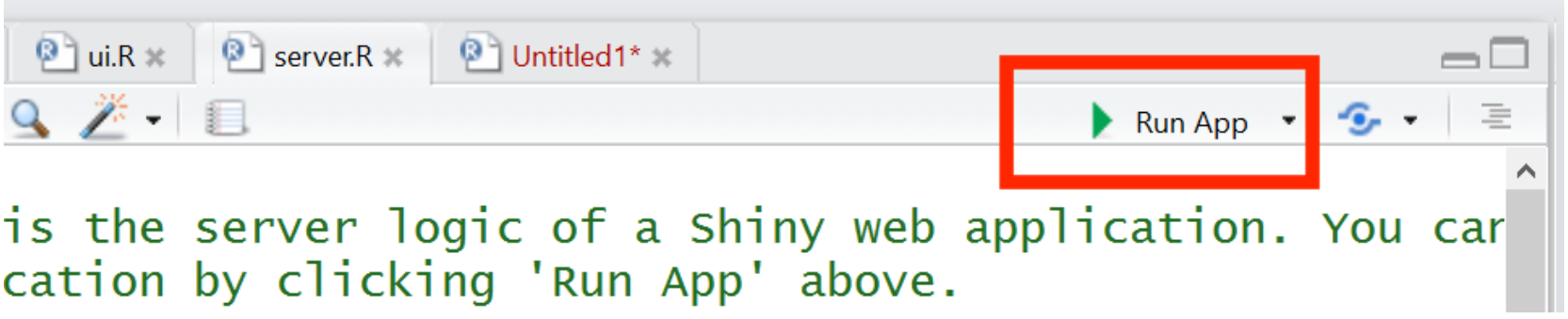

RStudio File Edit Code View Plots Session Build Debug Profile Tools Window Help
New File
New Project...
Open File... ⌘O
Reopen with Encoding...
Recent Files
Open Project...
Open Project in New Session...
Recent Projects
Import Dataset
Save ⌘S
Save As...
Save with Encoding...
Save All ⌥⌘S
Knit Document ⇧⌘K
Publish...
Print...
Close ⌘W
Close All ⇧⌘W
Close All Except Current ⌥⇧⌘W
Close Project
Quit Session...
R Script ⇧⌘N
R Notebook
R Markdown...
Shiny Web App...
Text File
C++ File
R Sweave
R HTML
R Presentation
R Documentation
Addins
README
/gallery/](https://shiny.rstudio.com/gallery/)
](https://www.showmeshiny.com)
oducts/shiny/shiny-user-showcase/](https://www.rstu
ontient 2 fichiers :
ce" : ce fichier définit l'apparence de la page web et les inputs
xplique comment, côté serveur, R doit se comporter et quels calculs

first app


New Shiny Web Application
Application name:
Name
Application type:
Single File (app.R)
Multiple File (ui.R/server.R)
Create within directory:
~/Desktop
Browse...
Shiny Web Applications
Create
Cancel

Launch the application

Once the project is created, click on the "Run app" button



or run `runApp('application_name')`*

```
str(formals(runApp))
```

```
#> Dotted pair list of 8
#>  $ appDir        : language getwd()
#>  $ port          : language getOption("shiny.port")
#>  $ launch.browser: language getOption("shiny.launch.browser", interactive())
#>  $ host          : language getOption("shiny.host", "127.0.0.1")
#>  $ workerId      : chr ""
#>  $ quiet         : logi FALSE
#>  $ display.mode  : language c("auto", "normal", "showcase")
#>  $ test.mode     : language getOption("shiny.testmode", FALSE)
```

for more control

Concept

A Shiny application contains 2 files :

- `ui.R` for "user interface": this file defines the appearance of the web page and the inputs
- `server.R`: this file explains how, on the server side, R must behave and which calculations it must perform to return as outputs according to user behavior

---

or a single file `app.R` containing both the ui and the server logic

Two-file version (ui.R and server.R)

```
#
# fichier ui.R
#
# [...]
#
library(shiny)

# Define UI for application that draws a histogram
shinyUI(fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
       sliderInput("bins",
                   "Number of bins:",
                   min = 1,
                   max = 50,
                   value = 30)
    ),

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

```
#
# fichier server.R
#
# [...]
#

library(shiny)

# Define server logic required to draw a histogram
shinyServer(function(input, output) {

  output$distPlot <- renderPlot({

    # generate bins based on input$bins from ui.R
    x    <- faithful[, 2]
    breaks <- seq(min(x), max(x), length.out = input$bins +
1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = breaks, col = 'darkgray', border =
'white')

  })

})
```

Defining the ui (with input and output)

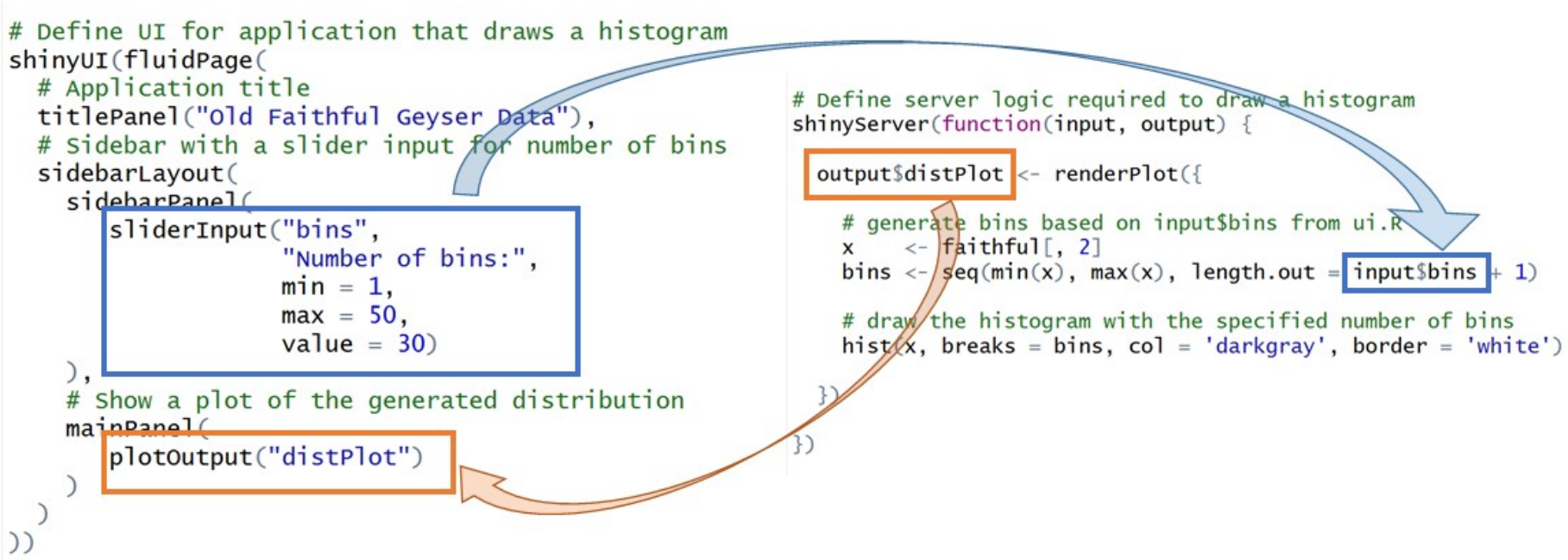
```
library(shiny)
shinyUI(fluidPage(
  # Application title
  titlePanel("Old Faithful Geyser Data"),
  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
        "Number of bins:",
        min = 1, max = 50, value = 30
      )
    ),
    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
))
```

Definition of server-side recipes (server.R)

```
library(shiny)

# Define server logic required to draw a histogram
shinyServer(function(input, output) {

  output$distPlot <- renderPlot({

    # generate bins based on input$bins from ui.R
    x    <- faithful[, 2]
    breaks <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = breaks, col = 'darkgray', border = 'white')

  })

})
```

Côté UI

Côté serveur



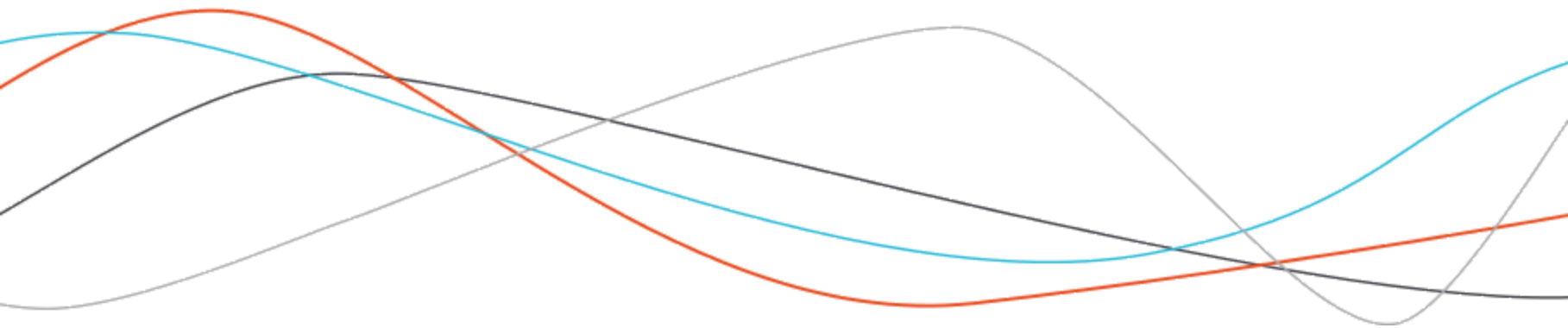
Mono-file app (app.R)

This is the recommended way to create simple shiny applications. The file must be called app.R.

```
library(shiny)

ui <- fluidPage(
  # code to define the ui.
  # what would go into ui.R
)

server <- function(input, output) {
  # server side
  # what would go into server.R
}

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

The Inputs

Interact with the user



inputs

```
ls( "package:shiny", pattern = "Input$" ) %>%
  grep( "^update", ., value = TRUE, invert = TRUE)
```

```
#>  [1] "checkboxGroupInput"      "checkboxInput"
#>  [3] "dateInput"               "dateRangeInput"
#>  [5] "fileInput"               "numericInput"
#>  [7] "passwordInput"           "restoreInput"
#>  [9] "selectInput"             "selectizeInput"
#> [11] "sliderInput"             "snapshotPreprocessInput"
#> [13] "textAreaInput"           "textInput"
#> [15] "varSelectInput"          "varSelectizeInput"
```

Buttons

```
ls( "package:shiny", pattern = "Button$")
```

```
#> [1] "actionButton"       "bookmarkButton"     "downloadButton"
#> [4] "modalButton"        "submitButton"       "updateActionButton"
```

Inputs

- The *inputs* are used to pass from information to R (*generally it is the user of the application who will do an action*)
- Each *input* is identified by a unique identifier
- They each have their own parameterization

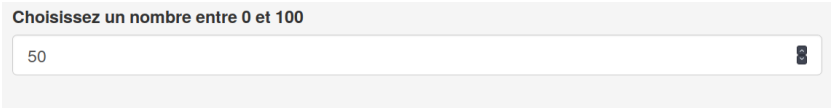
```
numericInput(
  # unique identifier, first parameter: inputId
  inputId = "Numeric",
  # label
  label = "Choose a number between 0 and 100",
  # parameters
  value = 50, min = 0, max = 100, step = 10
)
```

numericInput

A number

- Enter the value on the keyboard
- Use the arrows to increase/decrease the value

```
numericInput( "number",
  label = "Choose a number between 0 and 100",
  value = 50, min = 0, max = 100, step = 10
)
```

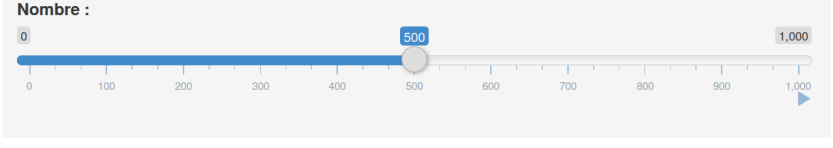


sliderInput

A number

- using a *slider*

```
sliderInput("slider1", "Number :",
  min = 0, max = 1000, value = 500, animate = TRUE, step = 10
)
```

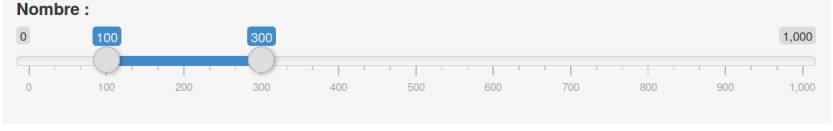


sliderInput

A number

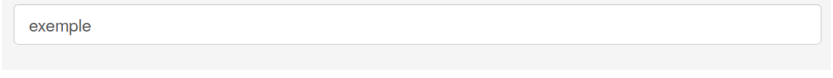
- or interval

```
sliderInput("slider2", "Number :",
            min = 0, max = 1000, value = c(100, 300)
)
```



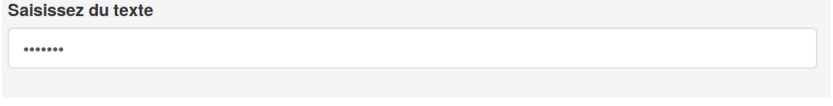
textInput

```
textInput(
  "texte",
  label = NULL, value = "example"
)
```



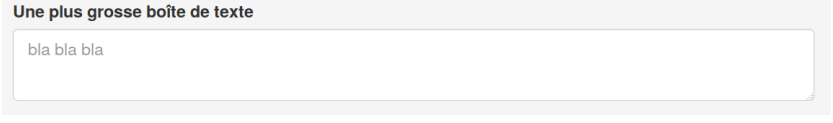
passwordInput

```
passwordInput(
  "password",
  label = "Saisissez du texte", value = "my_password"
)
```



textAreaInput

```
textAreaInput(
  "text",
  label = "bigger textArea", placeholder = "bla bla bla"
)
```

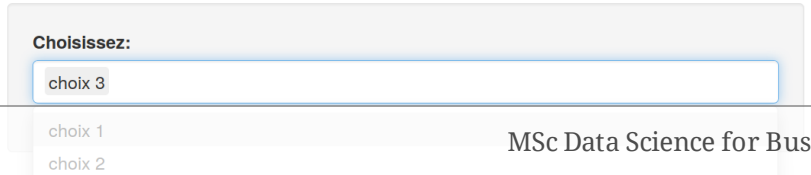


selectInput unique choice...

```
selectInput(
  "idSelect",
  label = "choose: ",
  selected = 3, choices = c("choice 1" = 1, "choice 2" = 2, "choice 3" =
3)
)
```

... or multiple

```
selectInput(
  "idSelect",
  label = "choose: ", selected = 3, multiple = TRUE,
  choices = c("choice 1" = 1, "choice 2" = 2, "choice 3" = 3)
)
```



checkboxInput: simple checkbox

```
checkboxInput(
  "check1",
  label = "Tick the box: ", value = TRUE
)
```

☑ Tick the box:

checkboxGroupInput: les checkbox multiples

```
checkboxGroupInput(
  "checkmultiple",
  label = "Tick (or not)", choices = c("check 1", "check 2", "check 3",
"check 4"),
  selected = "check 1"
)
```

Tick (or not)

- [x] check 1
- [ ] check 2
- [ ] check 3
- [ ] check 4

radio Buttons

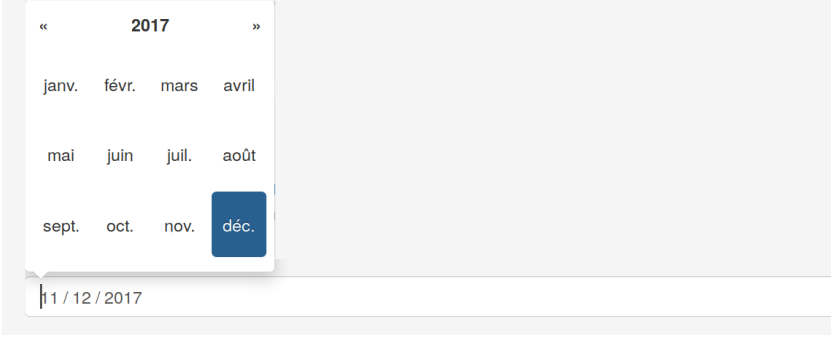
```
radioButtons("dist",
 label = "Distribution type:",
 choices= c(
    "Normal" = "norm",
    "Uniform" = "unif",
    "Log-normal" = "lnorm",
    "Exponential" = "exp"
  )
)
```

Distribution type:

- Normal
- Uniform
- Log-normal
- Exponential

dateInput: les dates

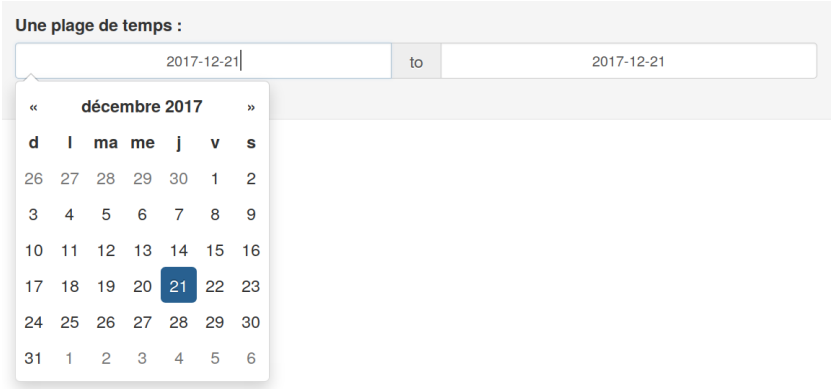
```
dateInput("ladate",
  label = "a date", value = Sys.Date() - 10, format = "dd / mm / yyyy",
  language = "fr", startview = "year",  weekstart = 1
)
```



dateRangeInput: date

ou une plage de temps :

```
dateRangeInput(
  "plagedate",
  label = "time interval : ", language = "fr"
)
```

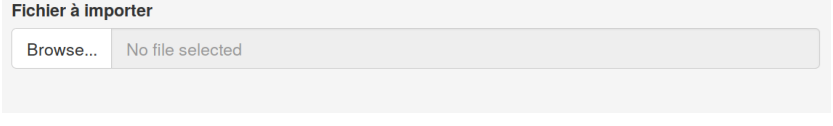


fileInput: one file or several files

`fileInput` allows you to choose a file on the client machine*

- the `multiple` argument allows the selection of several files

```
fileInput(
  "fichier_importe",
  label = "File to upload", multiple = FALSE,
  accept = c(
    "text/csv",
    "text/comma-separated-values,text/plain",
    ".csv"
  )
)
```



the `{shinyFiles}` package allows you to choose files on the server machine's file system

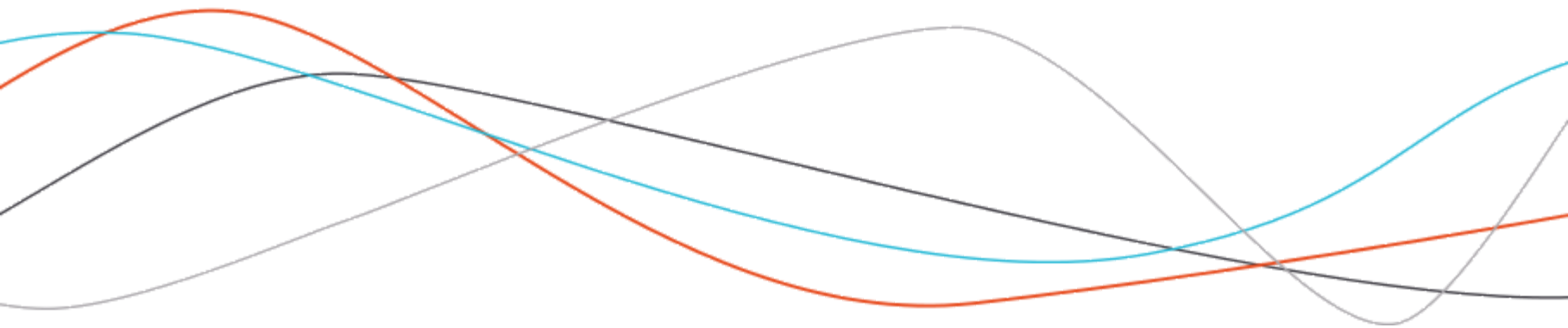
an action using a button

```
actionButton(
  "button1", label = "Go"
)
```

Go

Ouputs

What can be presented to the user



The different types of output

- The outputs are *spaces to be filled* by R
- Each output will be associated with a renderer in the server logic

```
ls( "package:shiny", pattern = "Output$" )
```

```
#> [1] "dataTableOutput"          "htmlOutput"
#> [3] "imageOutput"              "plotOutput"
#> [5] "snapshotPreprocessOutput" "tableOutput"
#> [7] "textOutput"               "uiOutput"
#> [9] "verbatimTextOutput"
```

textOutput & verbatimTextOutput

```
textOutput(outputId = "texte1")
verbatimTextOutput(outputId = "texte2")
```

Spoiler in the server part we make them like that:

```
output$texte1 <- renderText({
  # R code returning a string
})
```

ou

```
output$texte2 <- renderPrint({
  # R code returning console output
})
```

DToutput

```
library(DT)
DT::DTOutput(outputId ="table1")
```

Spoiler in the server part they are to be filled with `renderDT`*

```
output$table1 <- DT::renderDT({
  # a tibble or data.frame
})
```

In practice we will prefer the package `{DT}` and its functions `DTOutput` `renderDT` rather than `renderDataTable` and `DatatableOutput`.

plotOutput

```
plotOutput(outputId = "graphe")
```

Spoiler in the server part we render them with `renderPlot`.

```
output$graphe <- renderPlot({
  # ...
  ggplot(...) + ...
})
```

imageOutput

```
imageOutput(outputId = "image")
```

To return an image via `renderImage`.

htmlOutput & uiOuput

```
htmlOutput(outputId = "somehtml")
uiOutput(outputId = "uielement")
```

Spoiler in the server part we make them like that:

```
output$somehtml <- renderUI({
  ...
})
```

In the `renderUI` you can create components as in the ui part of the application, which allows for dynamic creation.**Tips** *Try to find another way to do it if possible,* `makeUI` *makes the code more complex to read*)

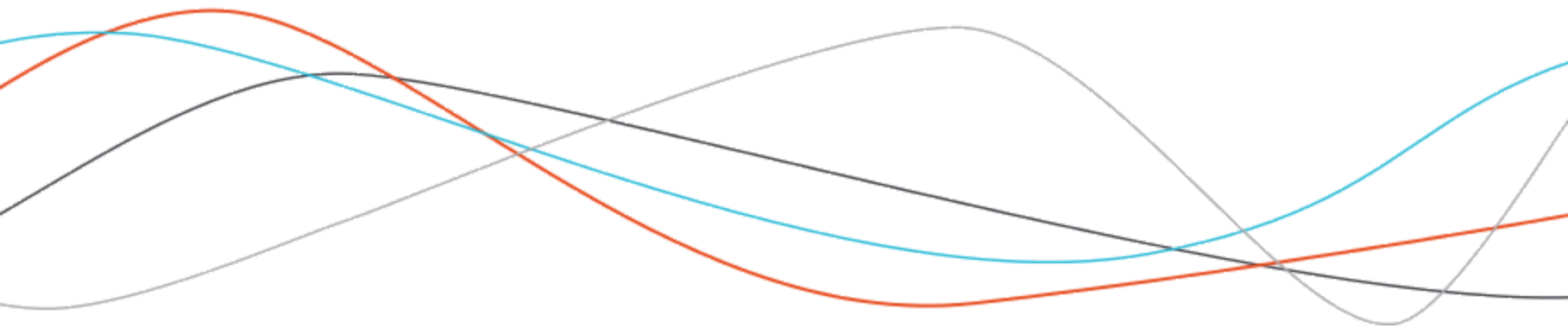
Overview

User Interface (ui.R)

- The functions of shiny interlock to build a ui
- The inputs and buttons to send information to R
- The outputs to create gaps to be filled by R

Shiny Serveur

the calculations that R must do

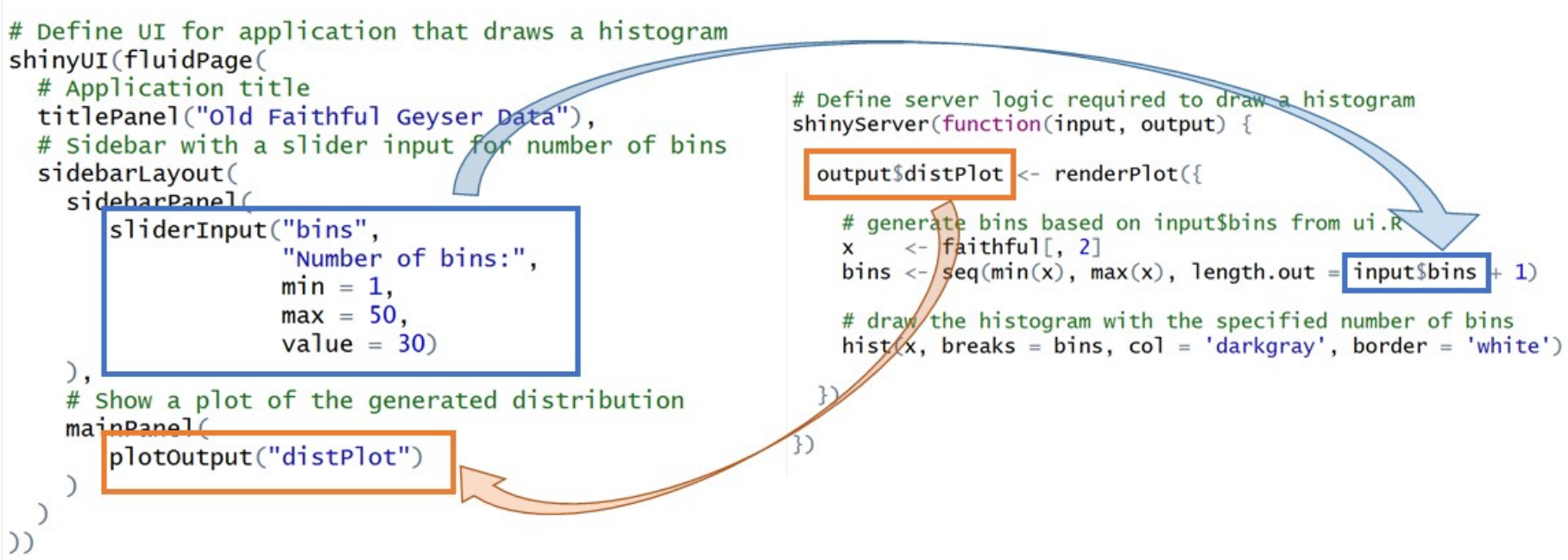


Server logic

- retrieve the input values by manipulating the `input$` object
- produce content for outputs with the object `output$`.
- understand the basic principles of responsiveness

Côté UI

Côté serveur



Instructions

The `server.R` file is a list of rules to "fill" the output spaces created by the ui.

```
shinyServer(function(input, output) {

  output$output1 <- render...({
    # R code using input$...
  })

  output$output2 <- render...({
    # R code using input$...
  })

})
```

Example

```
shinyServer(function(input, output){

  # The update rule for "plot1" output
  output$plot1 <- renderPlot({
    n <- input$slider1
    plot( rnorm(n), type = "l", lwd = 2)
  })

  # The update rule for "text1" output
  output$text1 <- renderPrint({
    n <- input$slider2
    rnorm(n)
  })

})
```

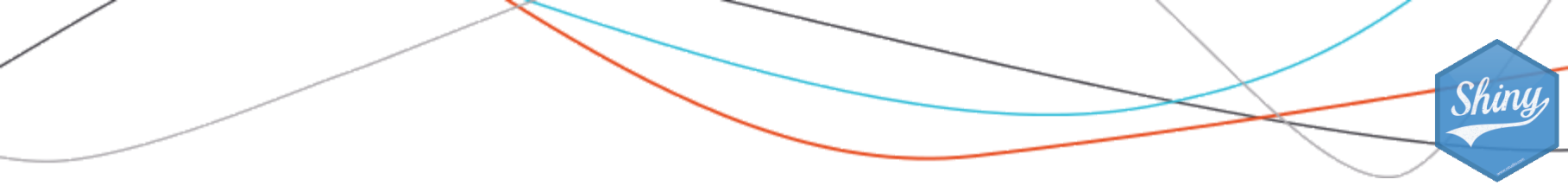
Shiny server

A shiny server is a function that is passed as a parameter to `shinyServer`.

```
shinyServer(function(input, output, session) {
  # R code using input & output
})
```

The content of this function uses:

- `input` to retrieve information from widgets created in the ui with the functions `...Input`
- `output` to declare how R updates the spaces left available by the functions `...Output`
- `session` is optional, it is used for *advanced* communication between the server and the client



The 3 server rules

Rule 1 : output$<output_name> <-

- We modify the outputs by manipulating the `output` object as a list
- We use an output only once
- We must imagine what we give to an output as a recipe to update it

```
# we modify the content of the output "xyz"
output$xyz <- ...

# forbidden !!
if( test == 1){
  output$abc <- ...
} else {
  output$abc <- ...
}
```

Rule 2 : output$<output_name> <- render...({...})

- To build the "recipe", we use a `render` function..., for example `renderPlot`, `renderText`...
- Always put the braces in the render*
- Inside the braces: the recipe, consisting of R instructions.

```
output$xyz <- renderPlot({
  # the R code which will produce a graph
  x <- rnorm(100)
  y <- rnorm(100)
  # old fashion scatter plot
  plot( x, y)
  })
```

Rule 3 : output$<output_name> <- render... ({ ... input$abc ... })

- In the recipe we use ingredients: the input
- We also manipulate the `input` object as a list, i.e. `input$abc`.

```
output$xyz <- renderPlot({
  #  we get the input$n ingredient that we then use in the recipe
  taille <- input$n
  x <- rnorm(taille)
  y <- rnorm(taille)
  plot( x, y )
})
```

Overview Rules 1, 2 & 3

```
shinyServer(function(input, output){

  output$plot1 <- renderPlot({
    n <- input$slider1
    plot( rnorm(n), type = "l", lwd = 2)
  })

})
```

- We use `output$xyz` to declare the recipe
- We use a `render...` function to create the recipe, we think of the braces!
- In the recipe, we use the ingredients, the `input$abc`.

All the renderer

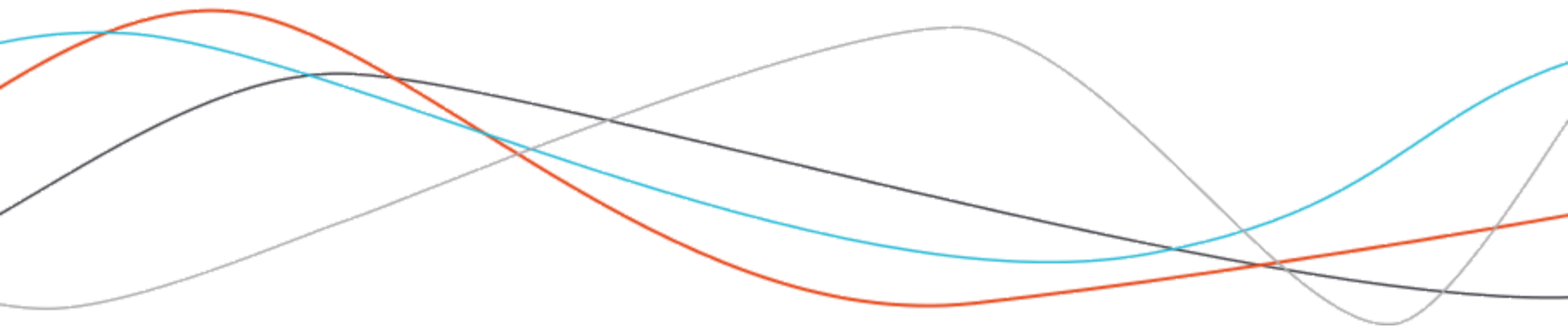
```
ls('package:shiny', pattern = "^render")
```

```
#> [1] "renderCachedPlot" "renderDataTable"  "renderImage"
#> [4] "renderPlot"       "renderPrint"      "renderTable"
#> [7] "renderText"       "renderUI"
```

They work *hand-in-hand* with the `*Output()` functions of the IU.

Shiny reactive

create reactiv objects



reactive - motivation

```
output$ex1_hist <- renderPlot({
  titre <- input$ex1_titre
  n <- input$ex1_n
  x <- rnorm(n) # <--------
  hist( x, main = titre, col="purple")
  rug( x )

})



output$ex1_info <- renderText({
  n <- input$ex1_n
  x <- rnorm(n) # <-------
  paste( "min = ", min(x) )
})
```

reactive

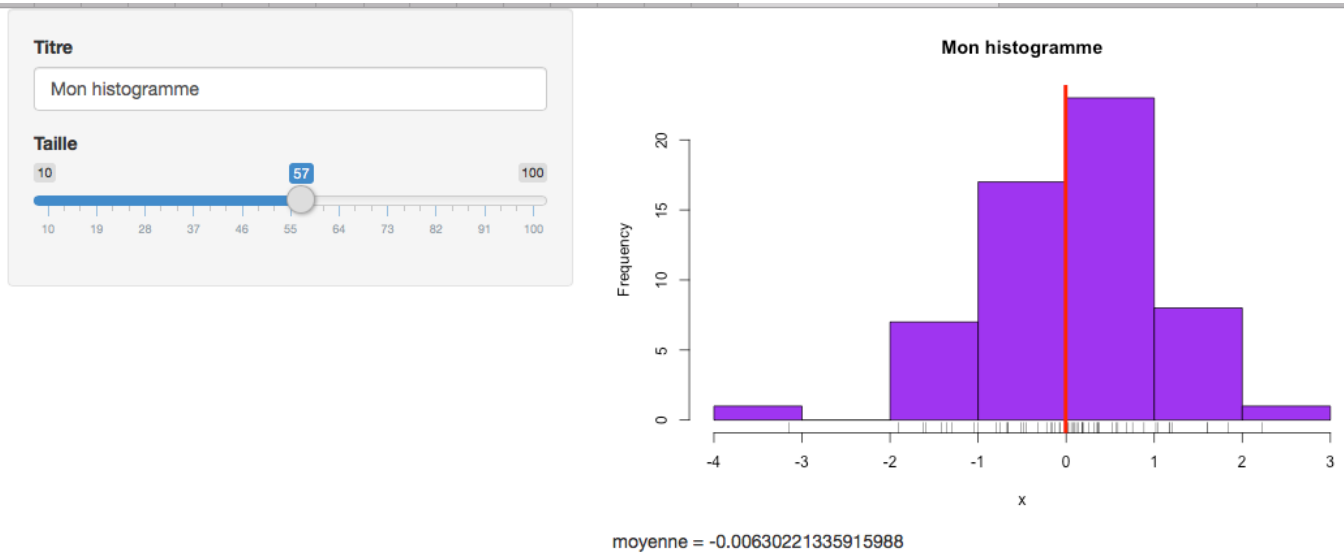
To create a reactive expression, we use the `reactive` function. We give him code that uses inputs (or something else that is also reactive):

```
data <- reactive({
  rnorm( input$n )
})
```

The created object is used as a function, without passing arguments to it. (be sure to never forget the `()` !)

```
x <- data()
```

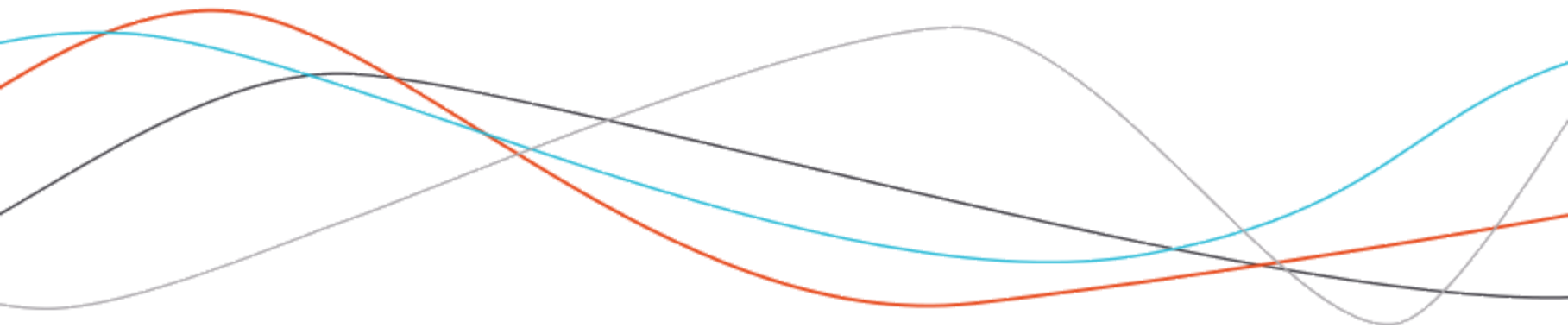
Exercice: Histogram + Mean



-> go to `ex1_reactive`

Shiny isolate

To prevent reactivity



Isolate: To prevent reactivity

`isolate` allows to recover the value of a reactive object without triggering the reactivity

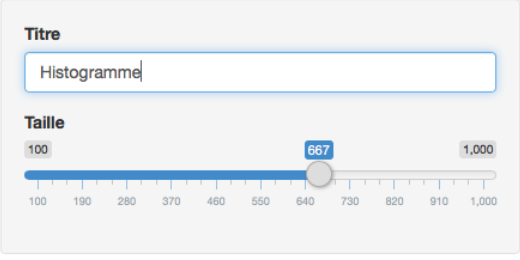
A `render` is, by default, updated as soon as one of the `input` used is modified by the user. To prevent an `input` from propagating reactivity, it should be placed between `isolate()`

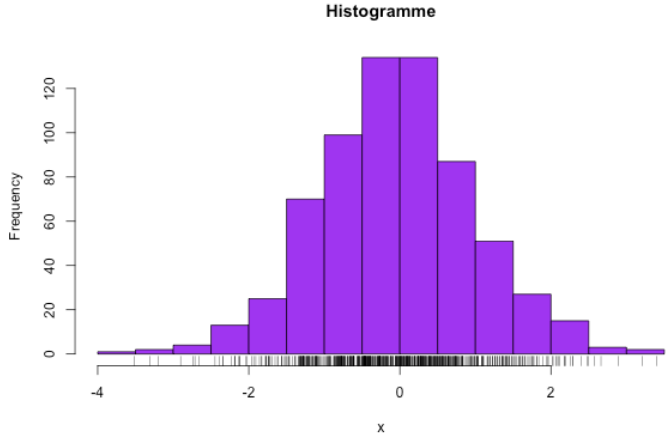
```
output$dessin <- renderPlot({

  create_graph( data = data(), couleur = isolate( input$couleur ), type =
intput$type )

})
```

Here changing the color will not regenerate the graph. While a modification of `data()` or `input$type` will restart the execution of the `create_graph` function.

isolate: Prevent reactivity

Exercise: make the graph reactive to the slider but not to the title field

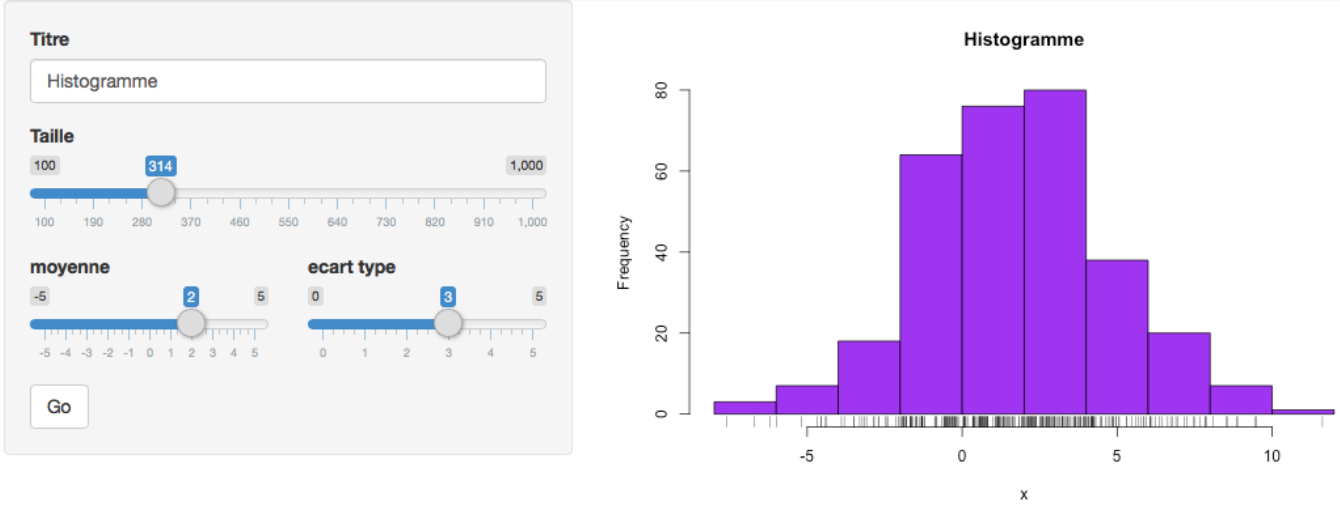




-> go to `ex2_isolate`

reactive + isolate

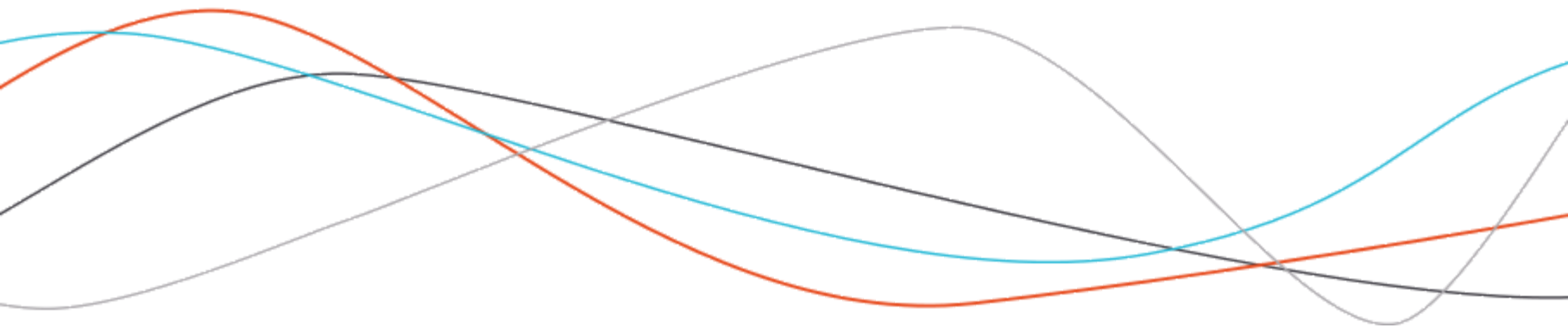
Using `reactive` and `isolate`, simulate the data only when the button is clicked



-> go to `ex3_reactive_isolate`

Shiny - eventReactive

Delay reactivity



eventReactive : Delay reactivity

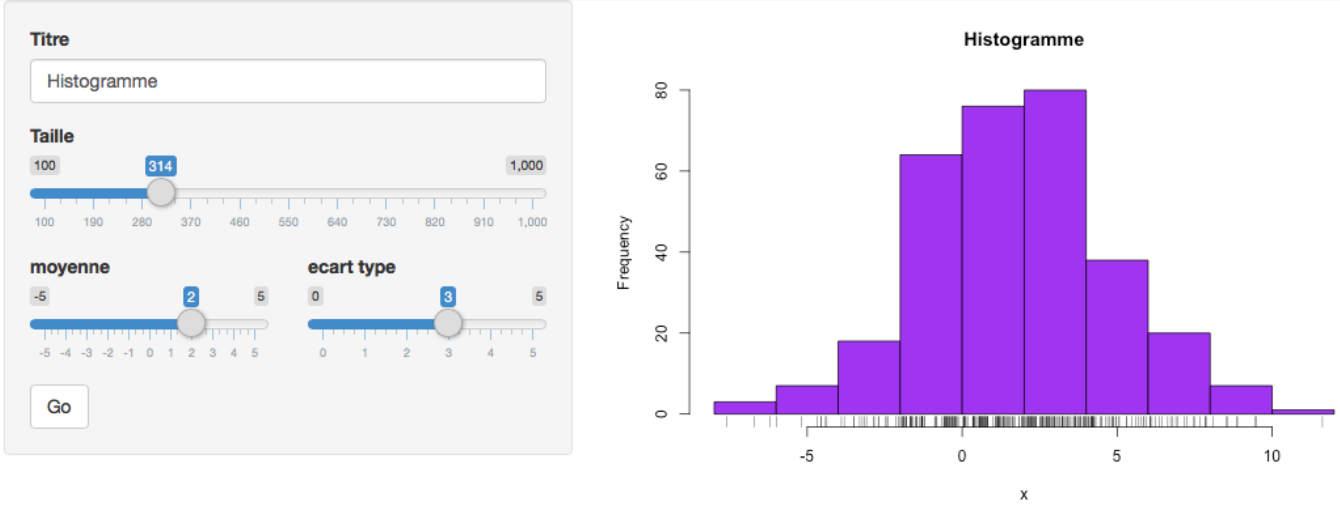
```
data <- eventReactive( input$go, {
  # of the code using other inputs or reactiv object
  # The code that appears here does not trigger reactivity
})
```

The code below is equivalent, but the intent is much less clear:

```
data <- reactive({
  input$go

  isolate({
  # of the code using other inputs or reactiv object
  # The code that appears here does not trigger reactivity
  })

})
```

eventReactive

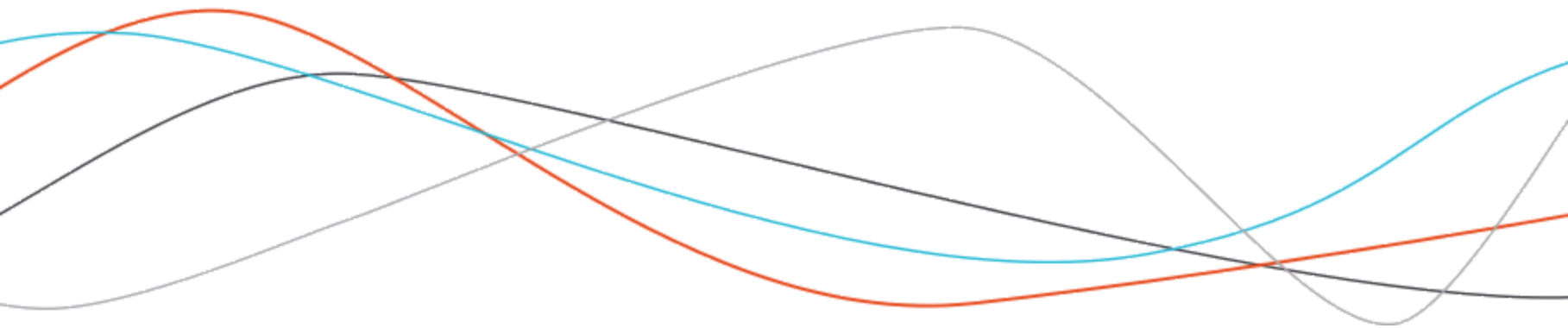
Repeat the previous exercise with `eventReactive`



-> go to `ex4_eventreactive`

Shiny observe

Control reactivity



observe

`observe` does not return anything, it is a function that allows to monitor a "reactive" instruction. The observation content is interpreted as soon as one of the reactive values (`input$`, `reactive()`,...) used in this content is updated.

```
observe({
  data()
  message("data change")
})
```

observeEvent

Similar to `observe` but the reactivity is specified.

```
observeEvent( input$n, {
  # an expression executed only when input$n
  # is invalidated, and even if it uses reactive values
})
```

We could imitate ObserveEvent with the couple `observe` / `isolate`:

```
observe({
  # no need to use `input$n`, it is enough that it is present in the
present code to trigger the reactivity
  input$n
  isolate({
# an expression executed only when input$n is invalidated
  })
})
```

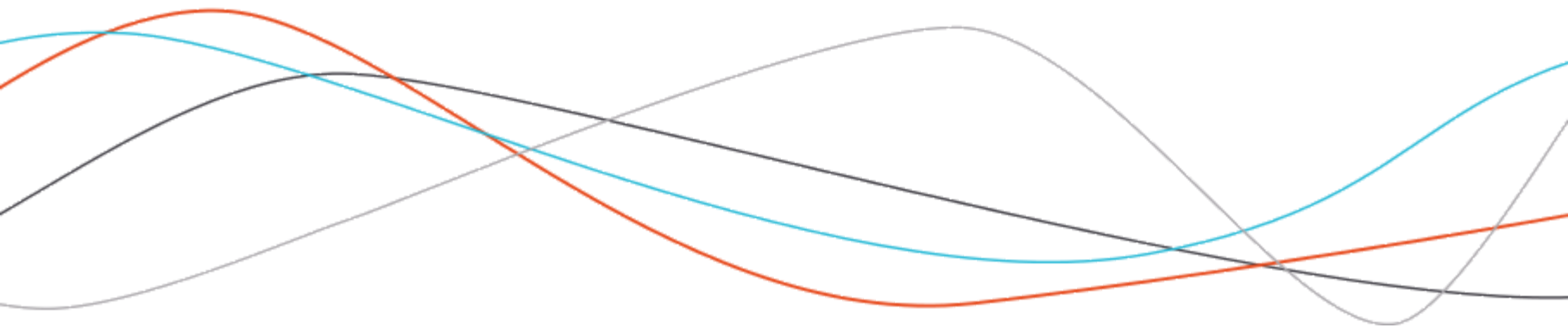
observe & observeEvent

Repeat the previous exercise, make sure that although the graph is not updated before the button is clicked, that the modification of one of the sliders generates a log message in the console such as "the size has been modified" or "the average has been modified".

-> go to `ex5_observe`

Agencement de l UI

arrange inputs and outputs



The interface consists of:

- Static parts: text
- Boxes to structure the content
- Inputs: buttons, lists,... to control what R does
- Outputs: outputs (graphs, tables,...) generated by R

Layout

The shiny functions build a graphical interface by generating html tags.

No need to know html, just the principle of boxes.

The shiny functions all work the same way

- The named arguments are used to configure the element
- Unnamed arguments are interlocking elements

Sidebar

```
sidebarLayout(
  sidebarPanel(
    ... # sidebar content
  ),
  mainPanel(
    ... # content at right
  )
)
```

Iris k-means clustering

X Variable

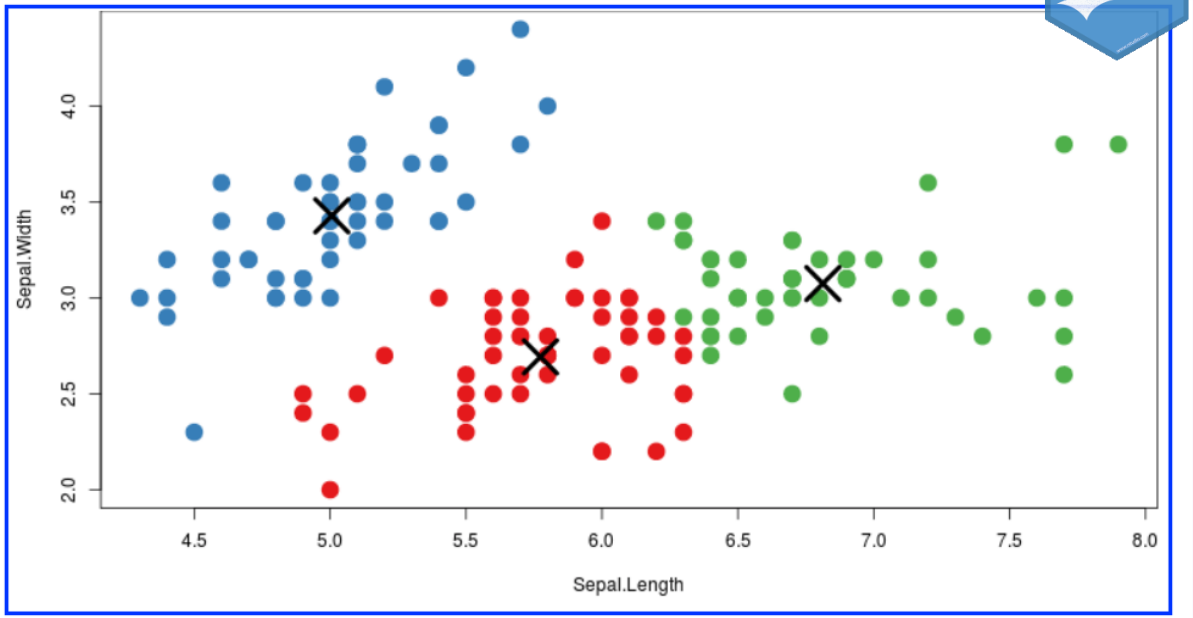
Sepal.Length

Y Variable

Sepal.Width

Cluster count

3



Kmeans example

by Joe Cheng <joe@rstudio.com>

server.R | ui.R

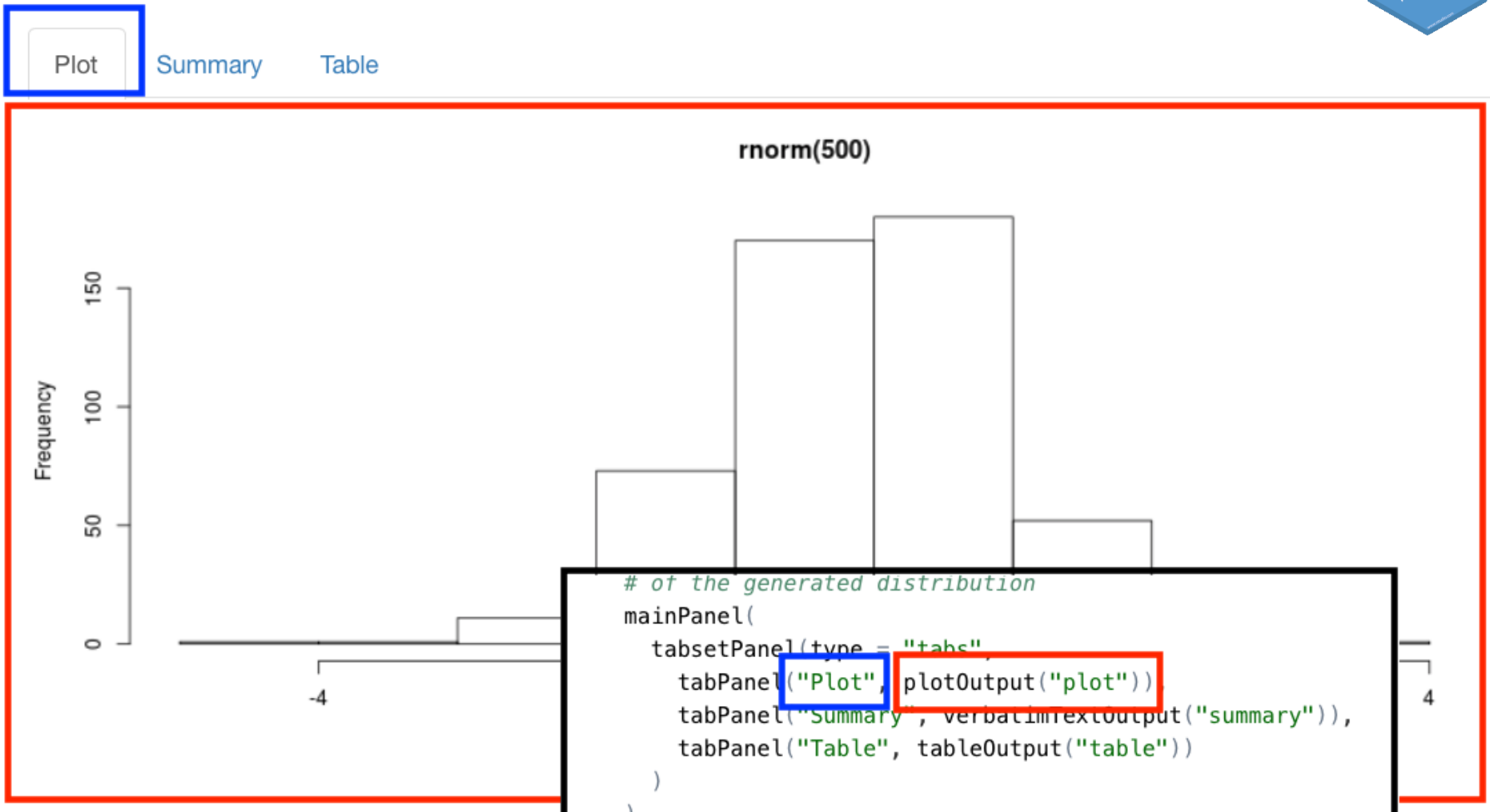
show with app

```
pageWithSidebar(
  headerPanel('Iris k-means clustering'),
  sidebarPanel(
    selectInput('xcol', 'X Variable', names(iris)),
    selectInput('ycol', 'Y Variable', names(iris),
                selected=names(iris)[[2]]),
    numericInput('clusters', 'Cluster count', 3,
                 min = 1, max = 9)
  ),
  mainPanel(
    plotOutput('plot1')
  )
)
```

Tabs

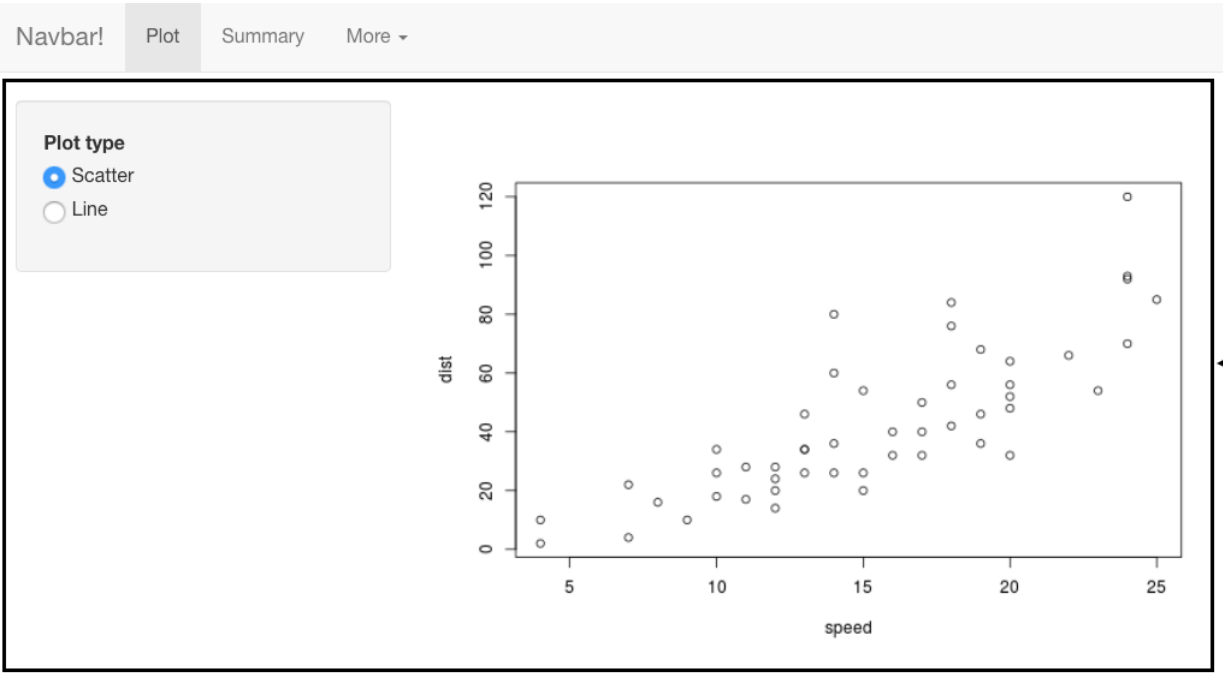
- `tabsetPanel` : a set of tabs
- `tabPanel` : a tab

```
tabsetPanel(
  tabPanel("tab 1", ... ),
  tabPanel("tab 2", ... )
)
```


Plot
Summary
Table
rnorm(500)
Frequency
0
50
100
150
-4
4
of the generated distribution
mainPanel(
tabsetPanel(type = "tabs",
tabPanel("Plot", plotOutput("plot")),
tabPanel("Summary", verbatimTextOutput("summary")),
tabPanel("Table", tableOutput("table"))
)
)
)

Navbar

```
navbarPage(
  tabPanel( "tab 1", ... ),
  navbarMenu( "menu 1",
    tabPanel( "sub menu 3-1", ... ),
    tabPanel( "sub menu 2-2", ... )
  )
)
```



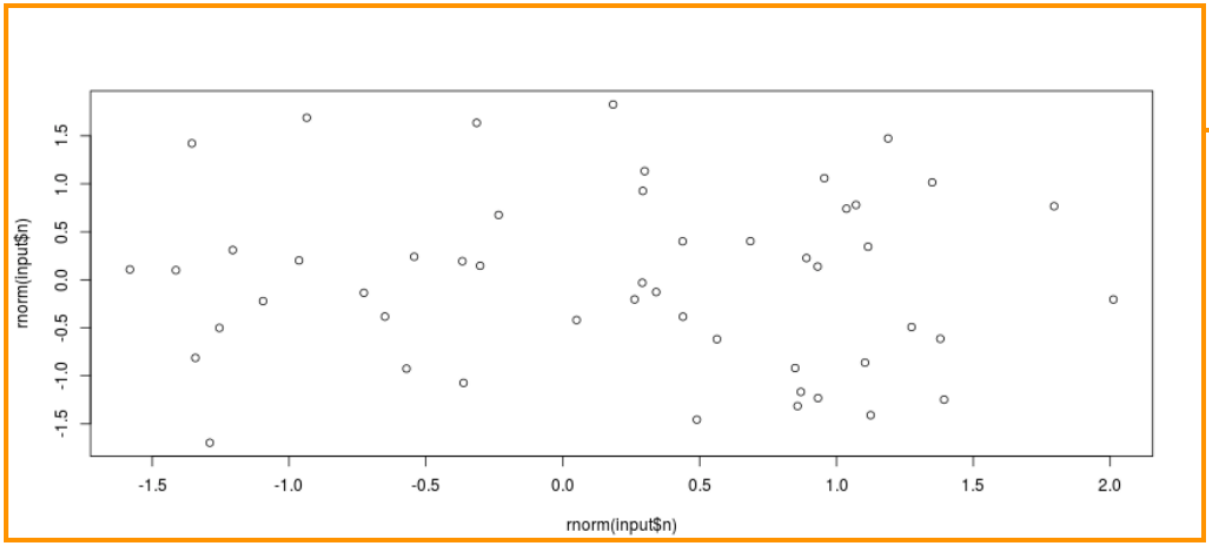
server.R ui.R

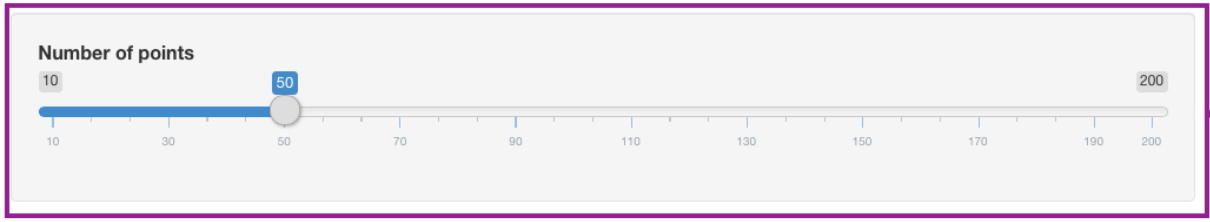
```
library(markdown)

navbarPage("Navbar!",
  tabPanel("Plot",
    sidebarLayout(
      sidebarPanel(
        radioButtons("plotType", "Plot type",
          c("Scatter"="p", "Line"="l")
        )
      ),
      mainPanel(
        plotOutput("plot")
      )
    )
  ),
  tabPanel("Summary",
    verbatimTextOutput("summary")
  ),
  navbarMenu("More",
    tabPanel("Table",
      DT::dataTableOutput("table")
    ),
    tabPanel("About",
```

Vertical layout

```
verticalLayout(
  # first box
  ... ,

  # second box ...
  ...
)
```

Vertical layout example



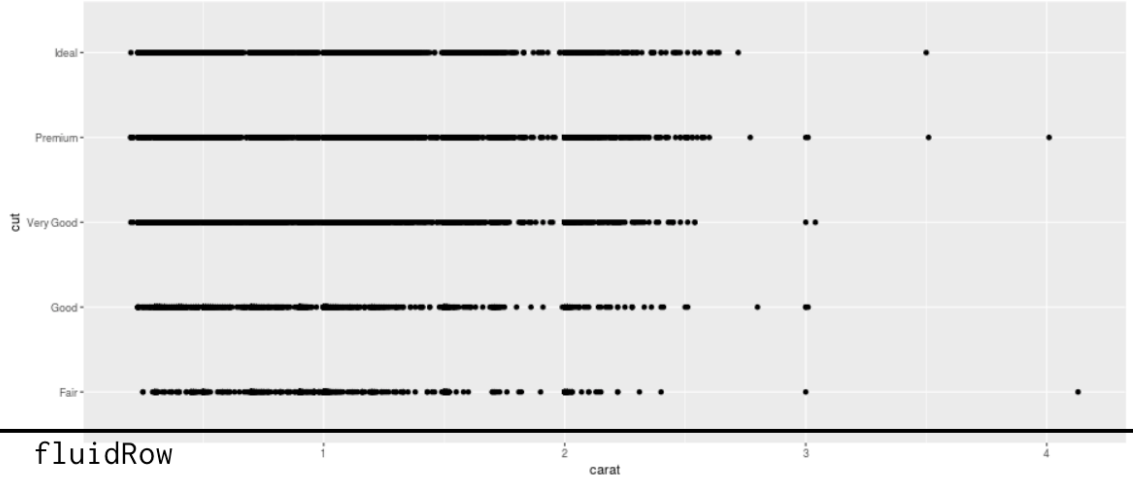


server.R | ui.R

```
fluidPage(
  verticalLayout(
    titlePanel("Vertical layout example"),
    plotOutput("plot1"),
    wellPanel(
      sliderInput("n", "Number of points", 10, 200,
                  value = 50, step = 10)
    )
  )
)
```

Colonnes

```
fluidRow(  # max 12 columns
  column(4, ... ), # a column of width 4
  column(2, ... ), # a column of width 2
  column(6, ... )  # a column of width 6
)
```



fluidRow

column(3, ...)

Diamonds Explorer

Sample Size

1 19,001 53,940

1 5,501 11,001 22,001 33,001 44,001 53,940

Jitter

Smooth

column(4, ...)

X

carat

Y

cut

Color

None

column(4, ...)

Facet Row

None

Facet Column

None

server.R ui.R

```
library(ggplot2)

dataset <- diamonds

fluidPage(

  title = "Diamonds Explorer",

  plotOutput('plot'),

  hr(),

  fluidRow(
    column(3,
      h4("Diamonds Explorer"),
      sliderInput('sampleSize', 'Sample Size',
                  min=1, max=nrow(dataset),
                  value=min(1000, nrow(dataset)),
                  step=500, round=0),
      br(),
      checkboxInput('jitter', 'Jitter'),
      checkboxInput('smooth', 'Smooth')
    ),
    column(4, offset = 1,
      selectInput('x', 'X', names(dataset)),
      selectInput('y', 'Y', names(dataset), names(dataset)[[2]]),
      selectInput('color', 'Color', c('None', names(dataset)))
    ),
    column(4,
      selectInput('facet_row', 'Facet Row',
        c(None='.', names(diamonds[sapply(diamonds, is.factor)]))),
      selectInput('facet_col', 'Facet Column',
        c(None='.', names(diamonds[sapply(diamonds, is.factor)])))
    )
  )
)
```

Absolute panel

```
absolutePanel(
  # choose where to position it
  top= , left = , right = , bottom = ,

  # ... its size
  width = 200, height = 200,

  # ... content
  ...

)
```

titre

... with tags h1, h2, h3, h4

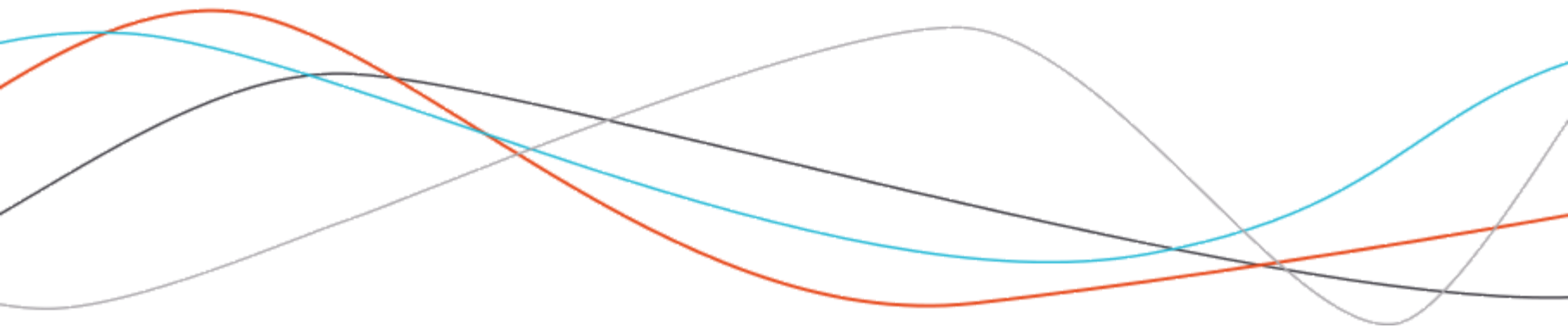
```
div(
  h1( "level 1"),
  h2( "level 2"),
  h3( "level 3"),
  h4( "level 4")
)
```

Static content, html, markdown, text

- `includeText` : include text file
- `includeHTML` : include html file
- `includeMarkdown` : include markdown

shiny exercise UI arrangement

**



Exercise: Reproduce this ui

titre niveau 1

lorem ipsum

chiffre 1: 10

chiffre 2: 20

titre niveau 2

chiffre 3: 10

chiffre 4: 20

chiffre 5: 10

chiffre 6: 20

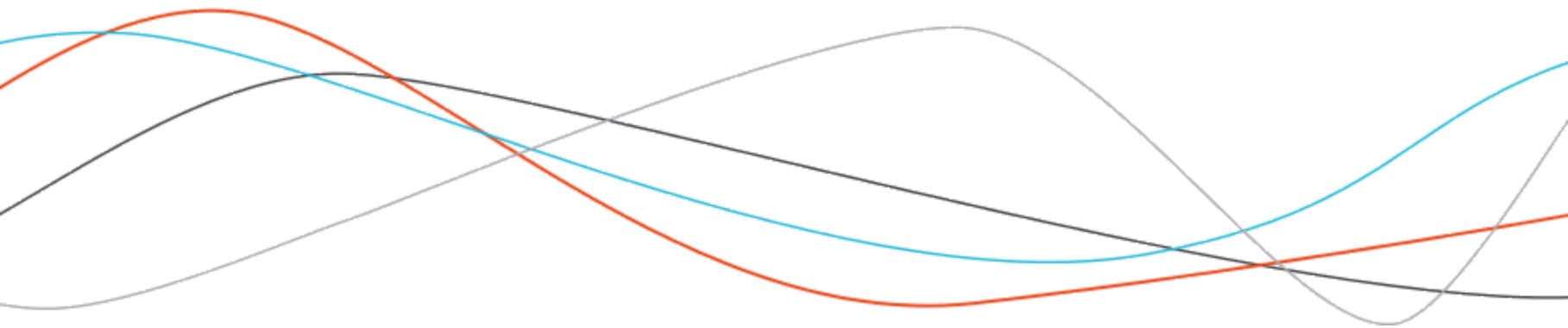
-> go to `ex6_ui`

Help: A classic sidebar layout

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

Shiny Update

Modify an UI



Updates functions

Each `Input` has a corresponding `update` function

```
ls( "package:shiny", pattern = "Input$" ) %>%
  grep( "^update", ., value = TRUE, invert = FALSE)
```

```
#>  [1] "updateCheckboxGroupInput" "updateCheckboxInput"
#>  [3] "updateDateInput"          "updateDateRangeInput"
#>  [5] "updateNumericInput"       "updateSelectInput"
#>  [7] "updateSelectizeInput"     "updateSliderInput"
#>  [9] "updateTextAreaInput"      "updateTextInput"
#> [11] "updateVarSelectInput"     "updateVarSelectizeInput"
```

Updates functions

The `update` allow to modify the characteristics of the IU.

```
ui <- fluidPage(
  selectInput("liste",label = "liste",choices = NULL),
  actionButton("go","init")
)
server <- function(input, output, session) {
  observeEvent(input$go,{
  updateSelectInput(session = session,
                    inputId = "liste",
                    choices=letters)})
}
shinyApp(ui, server)
```

This is a more elegant approach than the tedious use of `renderUI`. The server function **must be** defined with the `session` parameter.

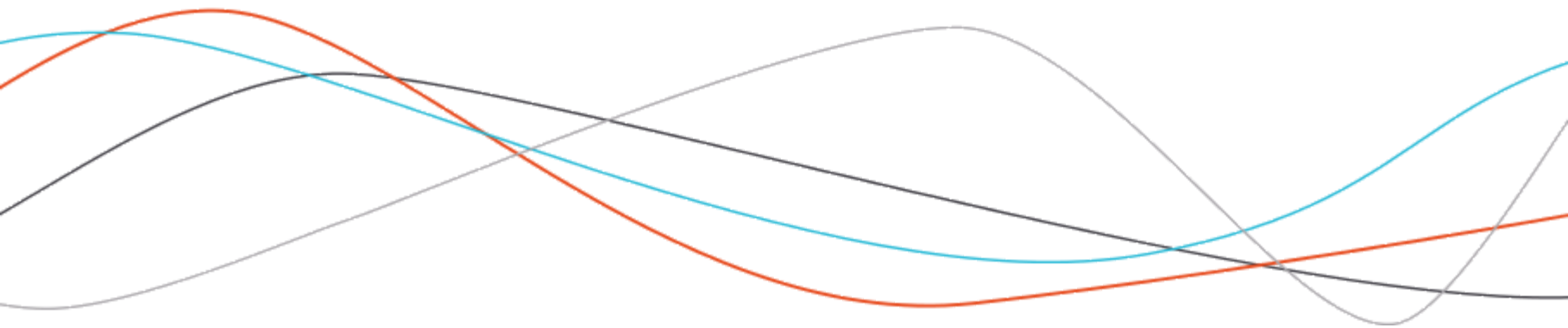
Exercise: Making a counter button

Each click on this button, must change the button label and indicate the number of times the button was clicked.

-> go to `ex11_update`

Shiny reactiveValues

store and modify reactive values



reactiveValues

The `reactiveValues` function creates a reactive list:

```
data <- reactiveValues(
  x = NULL
)
```

Which can then be modified in a reactive context:

```
data$x <- 3
```

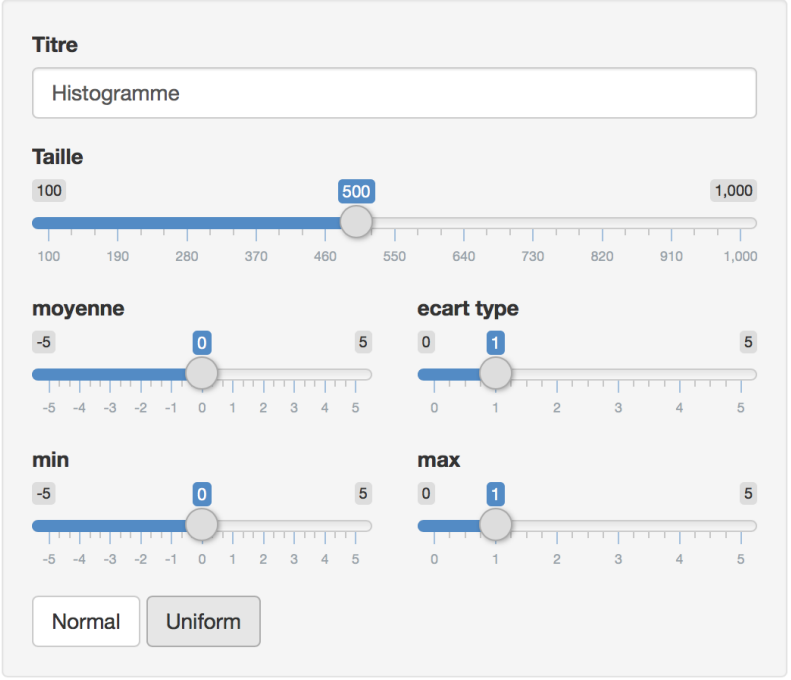
The value `x` of this list is invalidated, dependent reactive contexts will be executed again. *(i.e.: all `render`,`observe`,`reactive`... that use data$x will be re-run)*

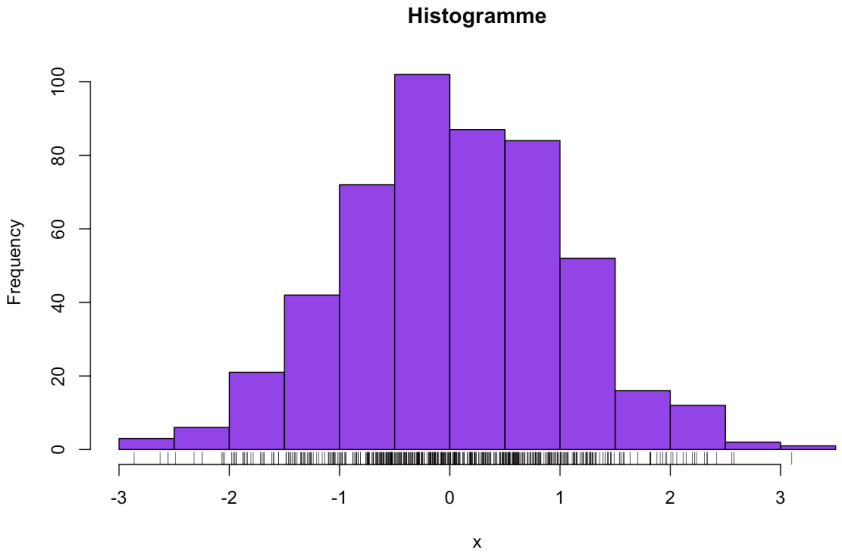
Exercise: Repeat the previous exercise (on `observe`) using this time the data in a `reactiveValues`.

This does not change the application for the moment. The only difference is that one the data is pre-existing before the first click on the button (ie: the first graph appears alone)

-> go to `ex7_reactivevalues`

Exercise : Normal or uniform according to the clicked button

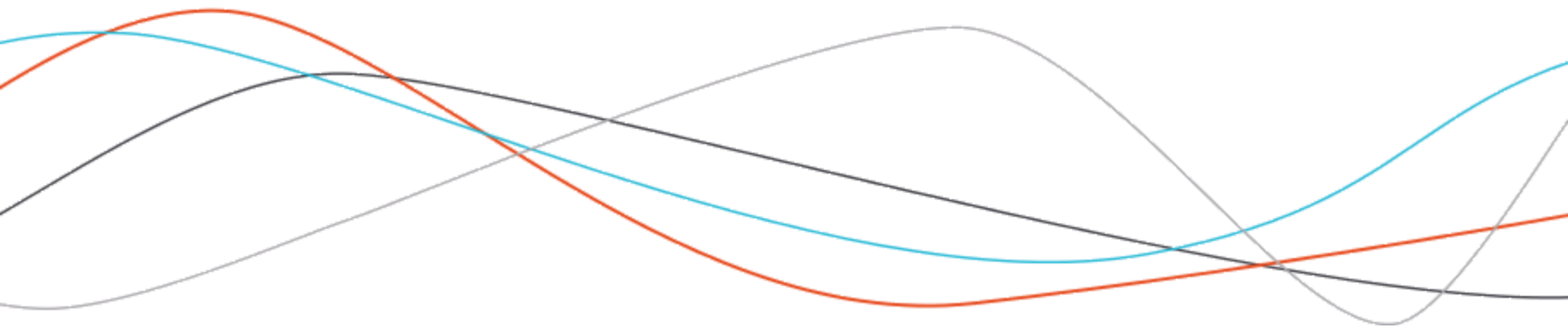




-> go to `ex8_reactivevalues_2`

Shiny conditionalPanel

Conditional display



Conditional display

conditionalPanel

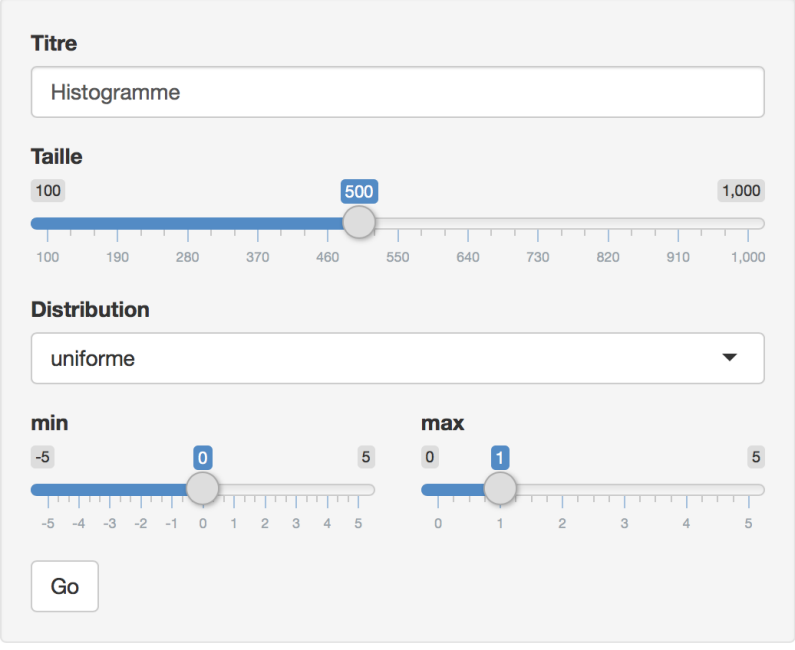
`conditionalPanel` allows to display its content conditionally to a javascript expression. *

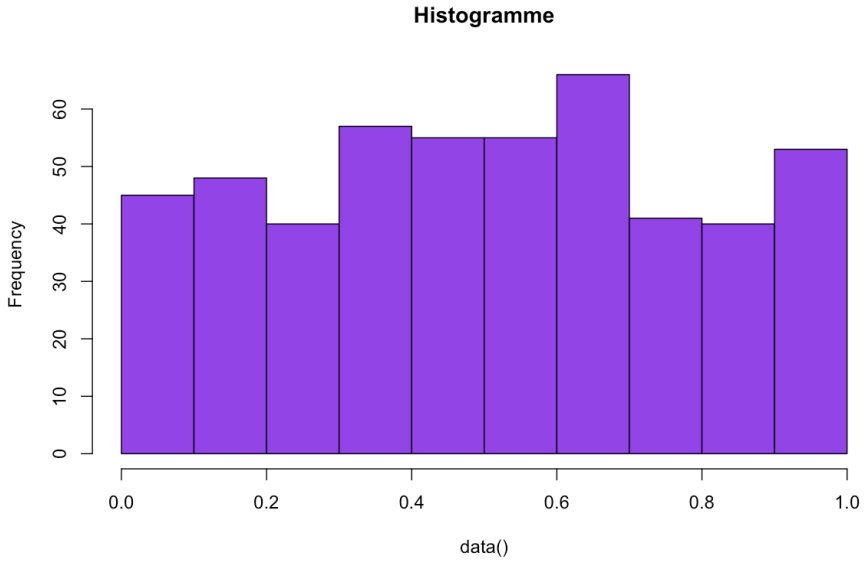
```
conditionalPanel('input.distribution == "normal"', {
  ...
})
conditionalPanel('input.distribution == "uniform"', {
  ...
})
```

Have a look to the package `{shinyjs}` which allows a more practical control of the display. see https://deanattali.com/shinyjs/overview#demo

replace the dollar with a point, for example `input$x` becomes `input.x` in the code javascript

Redo the previous application with two conditional panels

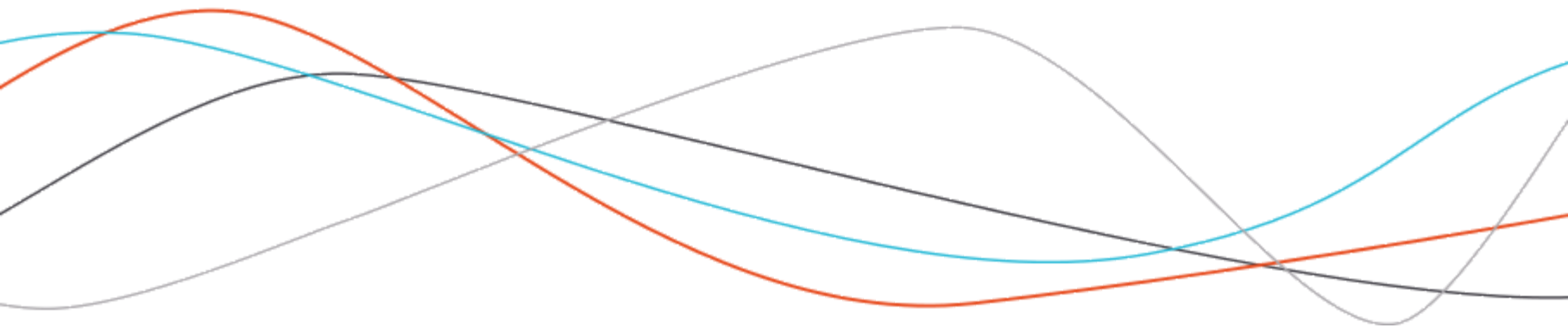




-> go to `ex9_conditionalPanel`

Shiny invalidateLater

Planned obsolescence



Planned obsolescence

invalidateLater

For the moment, we only react to user triggered events.

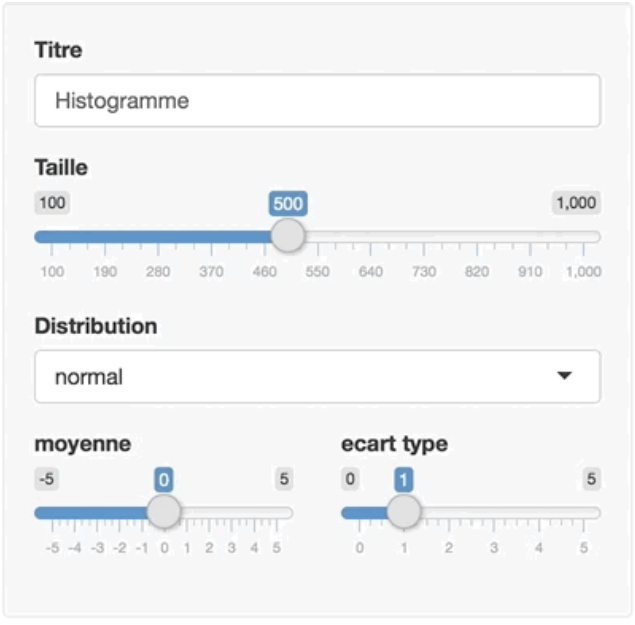
`invalidateLater` allows you to declare that the reactive context will become invalid.

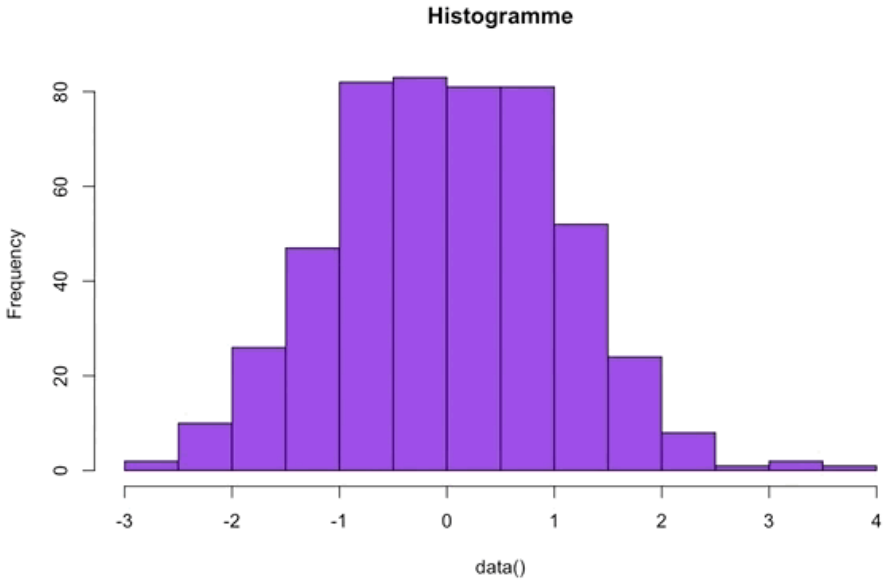
```
# invalidation dans 1 seconde
invalidateLater(1000)
```

A simple clock

```
ui <- fluidPage(textOutput("hour"))

server <- function(input, output, session) {
  output$hour <- renderText({
    invalidateLater(1000)
    as.character(Sys.time())
    })
}

shinyApp(ui, server)
```

Exercise: Renewal of data every second

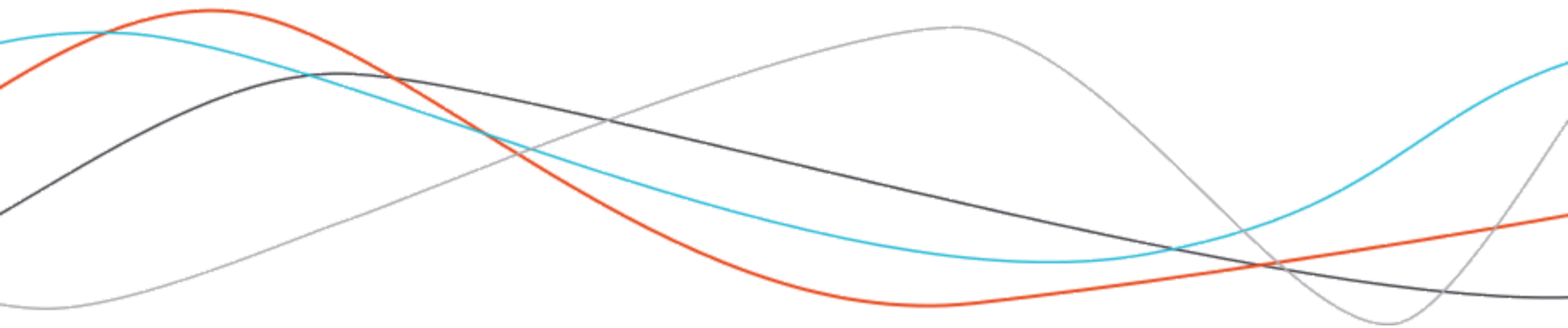




-> go to `ex10_invalidate`

Shiny Interaction with graphics

**



Interaction with graphics

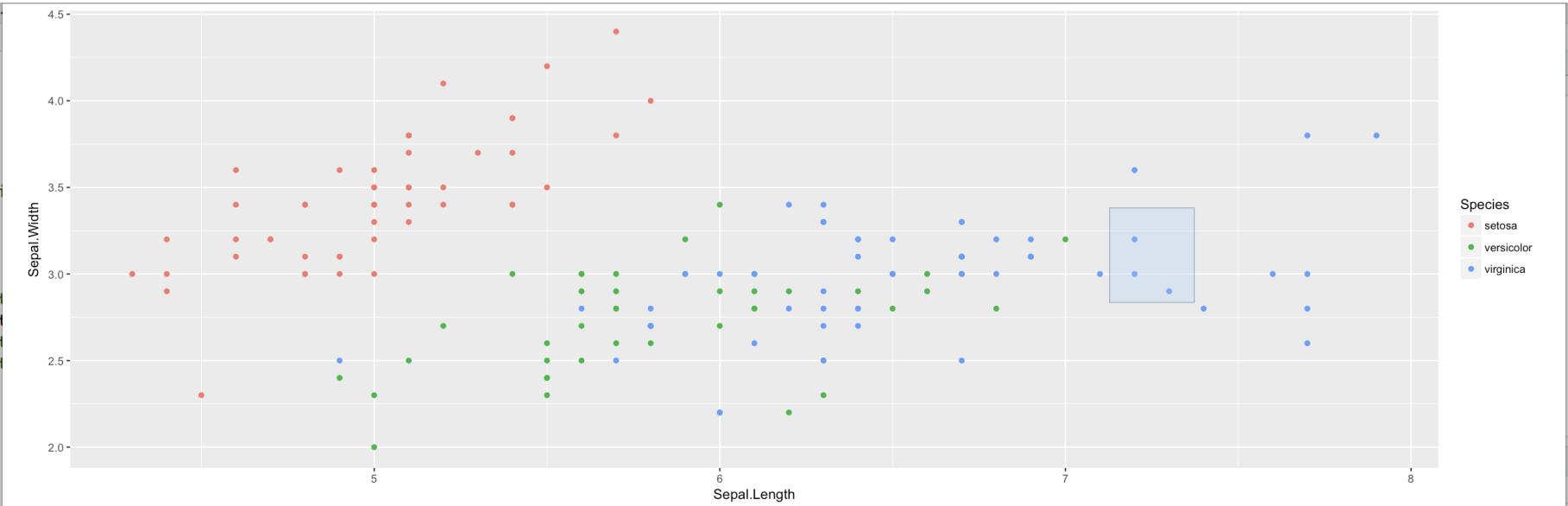
The ui `plotOutput` function allows you to declare click, dblclick, hover and brush event handlers

```
plotOuput( "plot",
  click = "plot_click", dblclick = "plot_dblclick",
  hover = "plot_hover", brush = "plot_brush"
)
```

We can then use (react to) in the server part:

- `input$plot_click`: when you click in the graph
- `input$plot_dblclick`: when you double-click in the graph
- `input$plot_hover`: when you fly over the graph
- `input$plot_brush` : when you make a selection in the graph

What do these objects contain?



input$click

```
$x
[1] 7.126008

$y
[1] 3.385602
```

input$dblclick

```
$x
[1] 6.244275

$y
[1] 3.414779
```

input$hover

```
NULL
```

input$brush

```
$xmin
[1] 7.126008

$xmax
[1] 7.371138
```

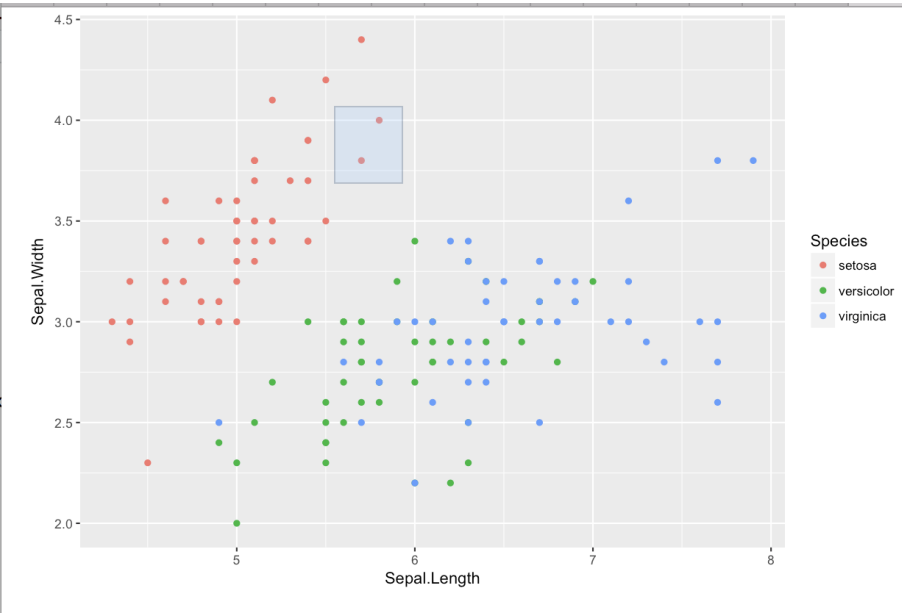
ui side

```
ui <- fluidPage(

  plotOutput("plot",
    click = "plot_click", dblclick = "plot_dblclick",
    hover = "plot_hover", brush = "plot_brush"
  ),

  splitLayout(
    div( h2("input$click"), verbatimTextOutput("txt_click" ) ),
    div( h2("input$dblclick"), verbatimTextOutput("txt_dblclick" ) ),
    div( h2("input$hover"), verbatimTextOutput("txt_hover" ) ),
    div( h2("input$brush"), verbatimTextOutput("txt_brush" ) )
  )

)
```

server side

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

  output$plot <- renderPlot({
    ggplot(iris, aes(Sepal.Length, Sepal.Width, color= Species) ) +
      geom_point()
  })

  output$txt_click     <- renderPrint({ input$plot_click })
  output$txt_dblclick  <- renderPrint({ input$plot_dblclick })
  output$txt_hover     <- renderPrint({ input$plot_hover })
  output$txt_brush     <- renderPrint({ input$plot_brush })

}
```

Display on the right of the graph a table of data selected in the graph



Show 10 entries Search:

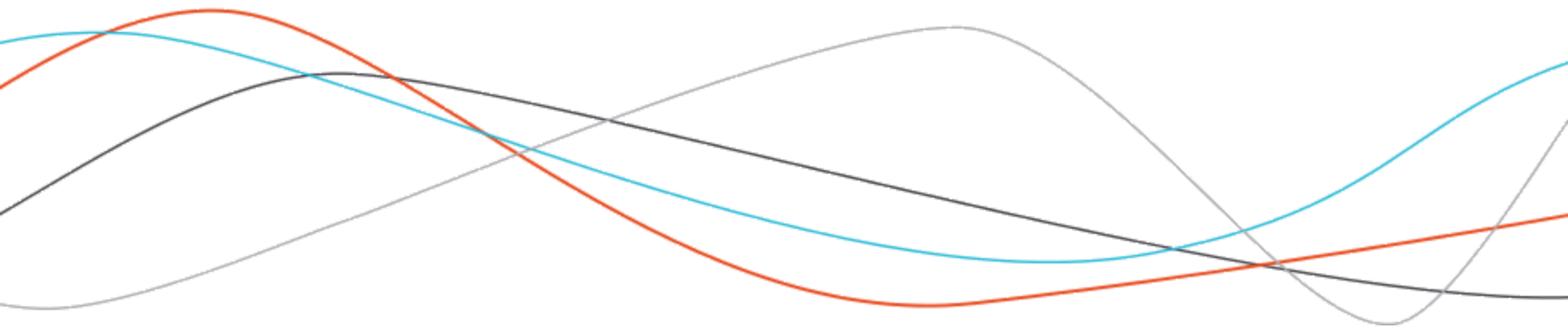
	Sepal.Length	Sepal.Width
1	5.8	4
2	5.7	3.8

Showing 1 to 2 of 2 entries Previous 1 Next

```
ui <- fluidPage(
  splitLayout(
    plotOutput("plot", brush = "plot_brush"),
    DT::DTOutput( "subset" )
  )
)
server <- function(input, output) {
  output$plot <- renderPlot({
    ggplot(iris, aes(Sepal.Length, Sepal.Width, color= Species) ) +
      geom_point()
  })
  data <- eventReactive(input$plot_brush, {
    brush <- input$plot_brush
    iris %>%
      select( Sepal.Length, Sepal.Width ) %>%
      filter(
        Sepal.Length > brush$xmin, Sepal.Length < brush$xmax,
        Sepal.Width > brush$ymin, Sepal.Width < brush$ymax
      )
  })
  output$subset <- DT::renderDT({
    DT::datatable(data())
  })
```

Shiny

Les modules shiny



The problem

In shiny, input and output must have a unique name within of the application.

It's manageable when you have a small application, but it becomes difficult to implement with a larger app.

When you want to reuse the same input/outputs set several times in the same application, you end up copying and pasting.

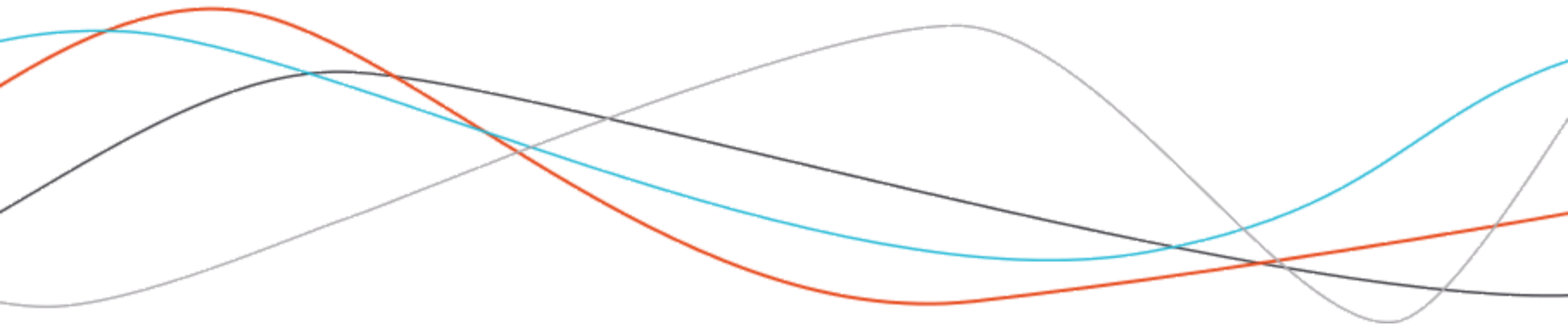
Shiny modules

We can see shiny modules as a shiny sub-application "self contained" to be integrated into a real application.

In a way, the modules are the shiny equivalent of the R functions.

They allow you to capture the repetition, manage the copy/paste in your place.

Gapminder Example



gapminder ... without modules - ui

```
ui <- fluidPage(
  tags$style(type="text/css", ".recalculating { opacity: 1.0; }"),
  titlePanel("Gapminder"),
  tabsetPanel(id = "continent",
    tabPanel("All",
      plotOutput("all_plot"),
      sliderInput("all_year", "Select Year", value = 1952, min = 1952,
        max = 2007, step = 5, animate = animationOptions(interval = 500))
    ),
    tabPanel("Africa",
      plotOutput("africa_plot"),
      sliderInput("africa_year", "Select Year", value = 1952, min = 1952,
        max = 2007, step = 5, animate = animationOptions(interval = 500))
    ),
    ...
  )
)
```

gapminder without modules - server

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

  # collect one year of data
  ydata_all <- reactive({
    filter(all_data, year == input$all_year)
  })

  ydata_africa <- reactive({
    filter(africa_data, year == input$africa_year)
  })
  # [...]
  output$all_plot <- renderPlot({

    # draw background plot with legend
    plot(all_data$gdpPercap, all_data$lifeExp, type = "n",
      xlab = "GDP per capita", ylab = "Life Expectancy",
      panel.first = {
      # [...]
      }
    )
  })
  # ...
  # output$africa_plot <- ...
  # output$americas_plot <- ...
}
```

gapminder ... with modules


```
library(shiny); library(dplyr)
source("data.R")
source("gapModule.R")

ui <- fluidPage(
  tabsetPanel(id = "continent",
    tabPanel("All", gapModuleUI("all")),
    tabPanel("Africa", gapModuleUI("africa"))
    # [...]
  )
)
server <- function(input, output) {
  callModule(gapModule, "all", all_data)
  callModule(gapModule, "africa", africa_data)
  # [...]
}
# Run the application
shinyApp(ui = ui, server = server)
```

Use a module

- The ui part

That's the easy part. Simply call the ui function of the module.

```
gapModuleUI("all")
```

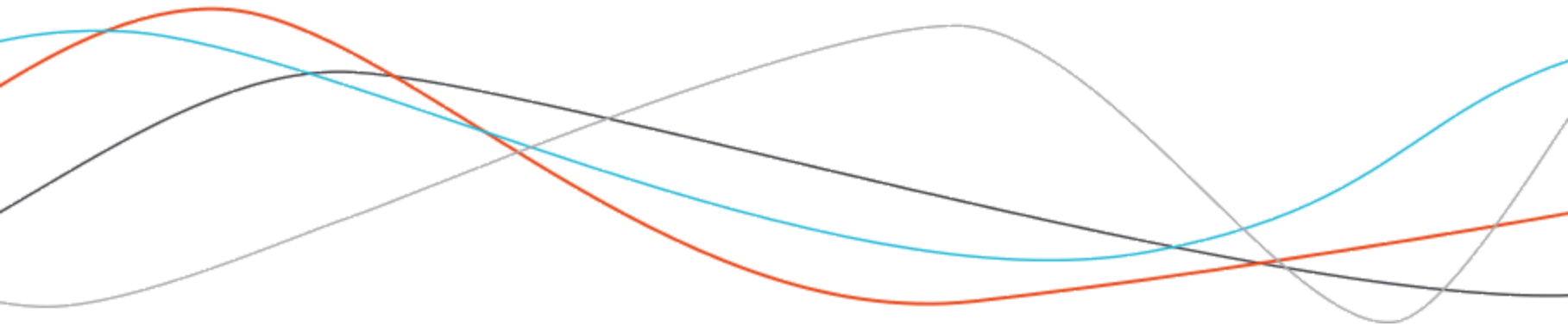
- The server part

We use the `callModule` function which takes as parameter:

- the server function
- the name of the module
- any parameters of the server function

```
callModule(gapModule, "all", all_data)
```

Creation of modules



The part ui

The ui function necessarily takes in first parameter `id`, it is the module name. The name given by the **user** of the module, not its developer.

This parameter is used to create a *namespace* `ns` function.

```
gapModuleUI <- function(id) {
  ns <- NS(id)
  # [...]
}
```

Then, each element (input, output,...) is named (the id argument) using the `ns` function. This time it is the developer of the module to put unique id names inside `ns`.

```
  tagList(
    plotOutput( ns("plot")),
    sliderInput( ns("year"), "Select Year", value = 1952,
      min = 1952, max = 2007, step = 5,
      animate = animationOptions(interval = 500)
    )
  )
```

Part ui: example

```
csvFileInput <- function(id, label = "CSV file") {
  ns <- NS(id)

  tagList(
    fileInput(ns("file"), label),
    checkboxInput(ns("heading"), "Has heading"),
    selectInput(ns("quote"), "Quote", c(
      "None" = "",
      "Double quote" = "\"",
      "Single quote" = "'"
    ))
  )

}
```

Server part

The server function of a module must take at least 3 arguments: input, output and session.

It looks like a shiny application server function.

```
gapModule <- function(input, output, session, data) {

  # [...]

  output$plot <- renderPlot({
    # [...]
  })
}
```

We use `output$` as in an application server function, except that the object `output` is scoped, you can only interact with the module outputs. `output$file` allows to access the `ns('file')`.

Same for `input$`, it only allows access to the module's inputs.

Server part: inputs

As input, after the 3 mandatory parameters (input, output, session), the module server functions can take other arguments, that will have been passed to it by `callModule`.

As in any R function, these parameters can be all and anything.

It is particularly interesting to pass *reactive functions* from in order to propagate reactivity in the application.

Server part: outputs

Similarly, a module server function can return an object. As usual this object can be a list if you want to return several things.

As with the entries, it is particularly interesting and idiomatic to return *a reactive function* as an output to propagate the reactivity manufactured in the module.

An example

```
csvFile <- function(input, output, session, stringsAsFactors) {

  userFile <- reactive({
    validate(need(input$file, message = FALSE))
    input$file
  })

  dataframe <- reactive({
    read.csv(userFile()$datapath,
      header = input$heading,
      quote = input$quote,
      stringsAsFactors = stringsAsFactors)
  })

  observe ({
    message( glue::glue("File {userFile()$name} uploaded"))
  })

  return(dataframe)
}
```

Exercise

Use `csvFileInput` and `csvFile` in a shiny application with as ui:

- a `csvFileInput` to select a csv file
- a `DT::DTOutput` to display the dataset

Another example: scatter plot linked

A ggplot function that requires a dataset (data) and a vector of column names (collars).

`data` contains the columns of `cols` and also the column `selected`

```
library(ggplot2)

scatterPlot <- function(data, cols) {
  ggplot(data, aes_string(x = colls[1], y = colls[2])) +
    geom_point(aes(color = selected_)) +
    scale_color_manual(values = c("black", "#66D65C"), guide = FALSE)
}
```

.... the ui function

2 studs using the same brush (`brush`)

```
linkedScatterUI <- function(id) {
  ns <- NS(id)

  fluidRow(
    column(6, plotOutput(ns("plot1"), brush = ns("brush"))),
    column(6, plotOutput(ns("plot2"), brush = ns("brush")))
  )

}
```

... the server function

```
linkedScatter <- function(input, output, session, data, left, right) {
  dataWithSelection <- reactive({
    brushedPoints(data(), input$brush, allRows = TRUE)
  })

  output$plot1 <- renderPlot({
    scatterPlot(dataWithSelection(), left())
  })

  output$plot2 <- renderPlot({
    scatterPlot(dataWithSelection(), right())
  })

  return(dataWithSelection)
}
```

.... the ui of the application

Simple, Basic

```
ui <- fluidPage(
  linkedScatterUI("plots")
)
```

... The application server

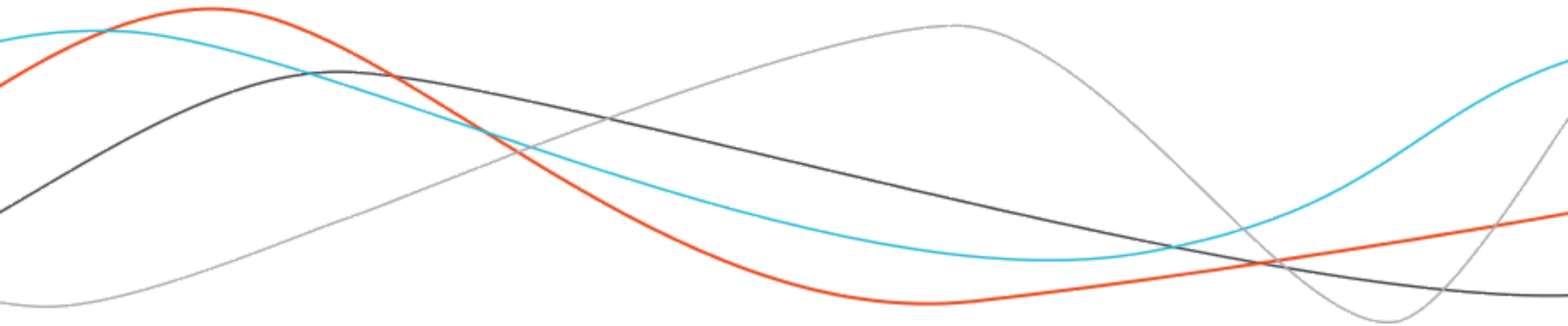
```
server <- function(input, output) {

  # we'll see later why these are functions
  left <-- function(){
    c("Sepal.Length", "Sepal.Width")
  }

  right <-- function(){
    c("Petal.Length", "Petal.Width")
  }

  data <- function(){
    iris
  }

  # module invocation
  selection <- callModule( linkedScatter, "plots", data, left, right)
}
```

Exercises:

- Show the selected points in a `DT::DTOutput`.
- Associate two `selectInput` to each plot to choose the columns instead to have them hard in the `left` and `right` functions
- Use a `csvFile` for `data`.
- Change the ui function so that you can choose where to inject the plotOutput.

Distribution

Where to put the modules?



app.R.

Simply put the module functions before creating des ui and server of the app.

We can possibly put them in a separate file that we `source`.

ui.R / server.R

If we opt for an application in two files, then the modules are to be defined in `globals.R`.

Otherwise, in a package

The best thing is to put the two functions of the module in one package.

Advantages:

- This encourages us to document them
- We'll tend to write code more to be used in other applications.

see https://rtask.thinkr.fr/blog/our-shiny-template-to-design-a-prod-ready-app/

Diane Beldame

diane@thinkr.fr

06 23 83 10 61