

Présentation R - Shiny, Société Générale

B.Thieurmél, benoit.thieurmél@datastorm.fr

25/10/2016

1 Shiny : créer des applications web avec le logiciel R

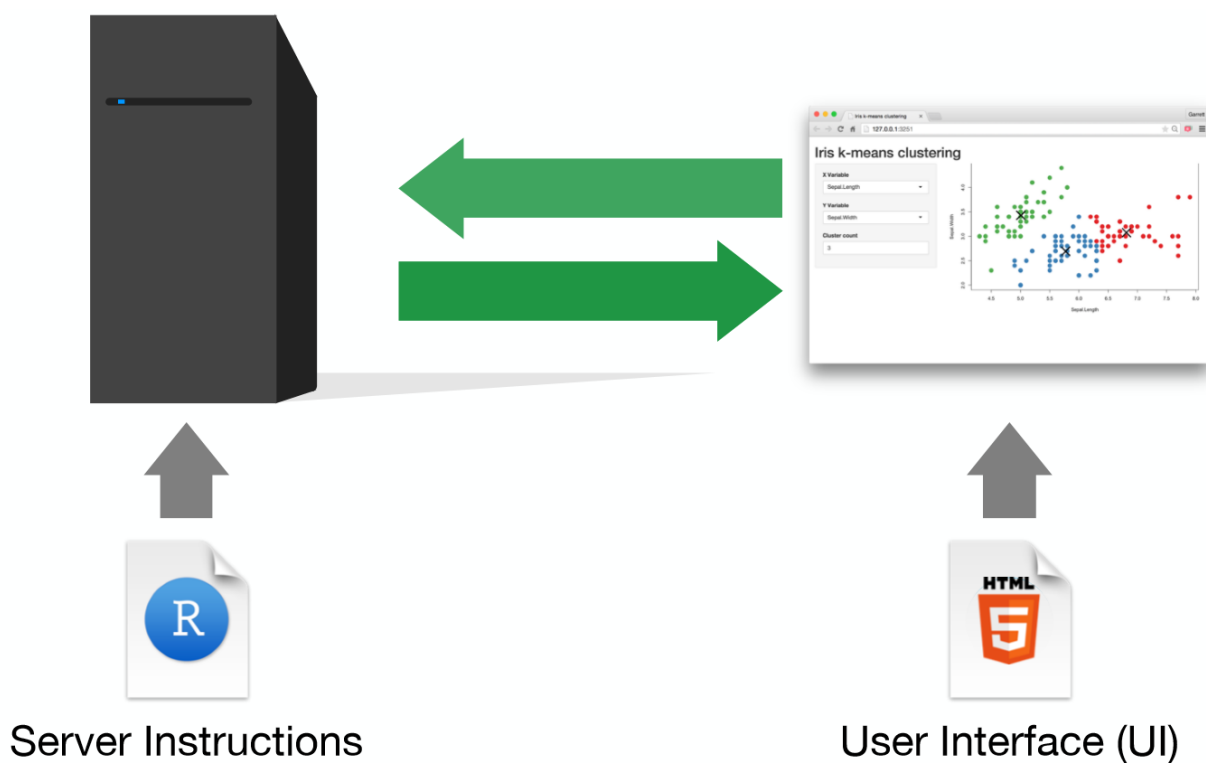
Shiny est un package **R** qui permet la création simple d'applications web interactives depuis le logiciel open-source **R**.

- pas de connaissances *web* nécessaires
- le pouvoir de calcul de R et l'interactivité du web actuel
- pour créer des applications locales
- ou partagées avec l'utilisation **shiny-server**

Plus de détails sur **Shiny** <http://shiny.rstudio.com>.

Plus de détails sur l'utilisation de **shiny-server** : <https://www.rstudio.com/products/shiny/shiny-server/>.

Une application **shiny** nécessite un ordinateur/un serveur exécutant **R**



© CC 2015 RStudio, Inc.

Figure 1: server

2 Ma première application avec shiny

- Initialiser une application est simple avec **RStudio**, en créant un nouveau projet
- File > New Project > New Directory > Shiny Web Application
- Basée sur deux scripts : ui.R et server.R
- Et utilisant par défaut le sidebar layout
- Commandes utiles :
- lancement de l'application : bouton **Run app**
- actualisation : bouton **Reload app**
- arrêt : bouton **Stop**

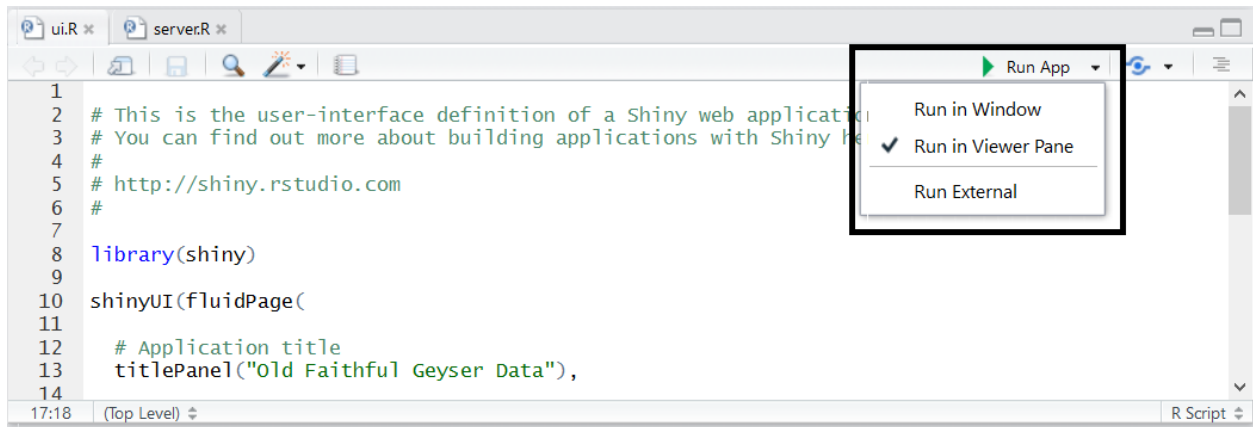


Figure 2: run

- **Run in Window** : Nouvelle fenêtre, utilisant l'environnement **RStudio**
- **Run in Viewer Pane** : Dans l'onglet *Viewer* de **RStudio**
- **Run External** : Dans le navigateur web par défaut

3 Interactivité et communication

3.1 Introduction

ui.R:

```
library(shiny)

# Define UI for application that draws a histogram
shinyUI(fluidPage(
  # Application title
  titlePanel("Hello Shiny!"),
  # Sidebar with a slider input for the number of bins
```

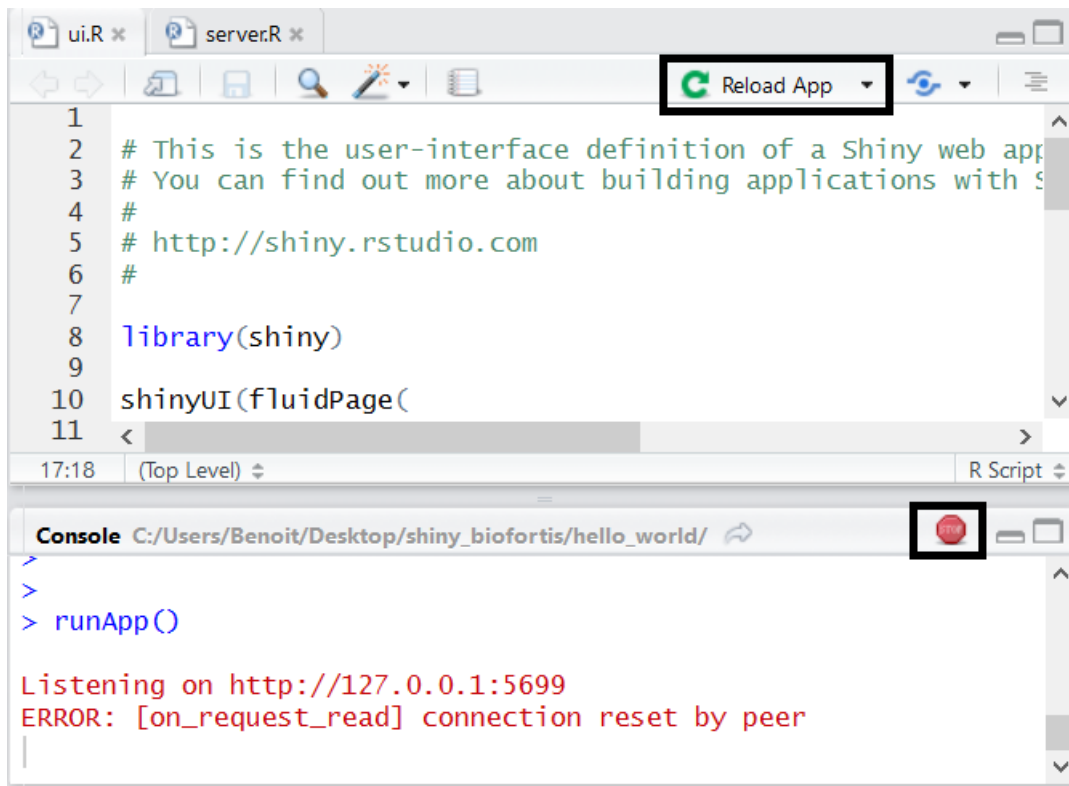


Figure 3: stop

```

sidebarLayout(
  sidebarPanel(
    sliderInput(inputId = "bins",
               label = "Number of bins:",
               min = 1, max = 50, value = 30)
  ),
  # Show a plot of the generated distribution
  mainPanel(plotOutput(outputId = "distPlot"))
)
))

```

server.R:

```

library(shiny)

# Define server logic required to draw a histogram
shinyServer(function(input, output) {
  # Expression that generates a histogram. The expression is
  # wrapped in a call to renderPlot to indicate that:
  #
  # 1) It is "reactive" and therefore should be automatically
  #    re-executed when inputs change
  # 2) Its output type is a plot
  output$distPlot <- renderPlot({
    x <- faithful[, 2] # Old Faithful Geyser data

```

```

bins <- seq(min(x), max(x), length.out = input$bins + 1)
# draw the histogram with the specified number of bins
hist(x, breaks = bins, col = 'darkgray', border = 'white')
})
})

```

Hello Shiny!

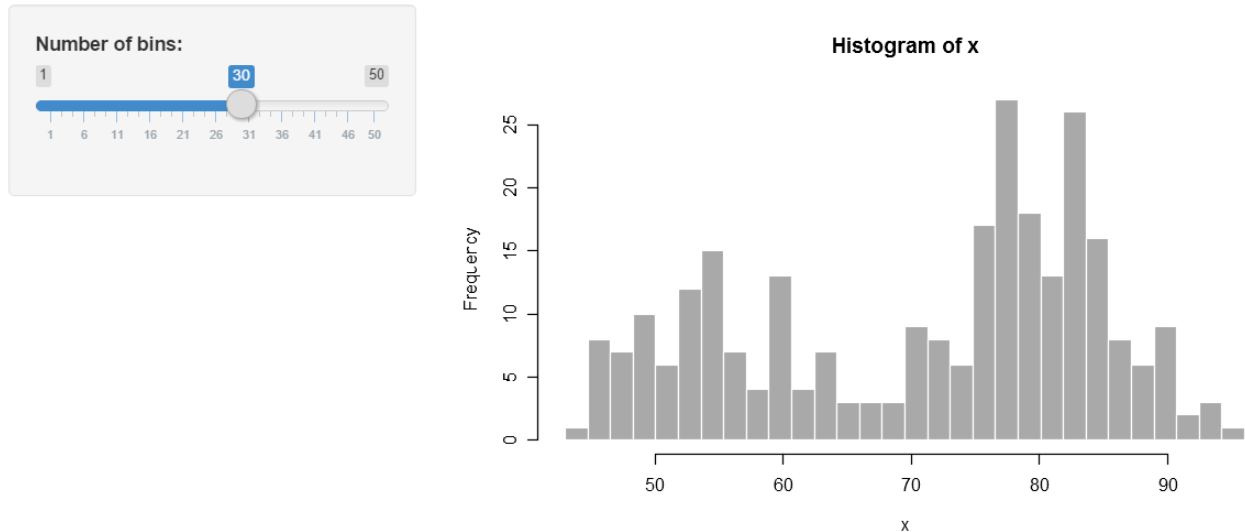


Figure 4: hello

Avec cette exemple simple, nous comprenons :

- Côté **ui**, nous définissons un slider numérique avec le code “`sliderInput(inputId = "bins", ...)`” et on utilise sa valeur côté **server** avec la notation “`input$bins`” : c’est comme cela que le **ui** créé des variables disponibles dans le **server** !
- Côté **server**, nous créons un graphique “`output$distPlot <- renderPlot({...})`” et l’appelons dans le **ui** avec “`plotOutput(outputId = "distPlot")`”, c’est comme cela que le **server** retourne des objet à **ui** !

3.2 Process

Le server et l’ui communiquent uniquement par le biais des inputs et des outputs

Par défaut, un output est mis-à-jour chaque fois qu’un input en lien change

3.3 Notice

la définition de l’interface utilisateur : **UI**

- la déclaration des inputs
- la structure de la page, avec le placement des outputs

la partie serveur/calculs : **SERVER**

- la déclaration et le calcul des outputs

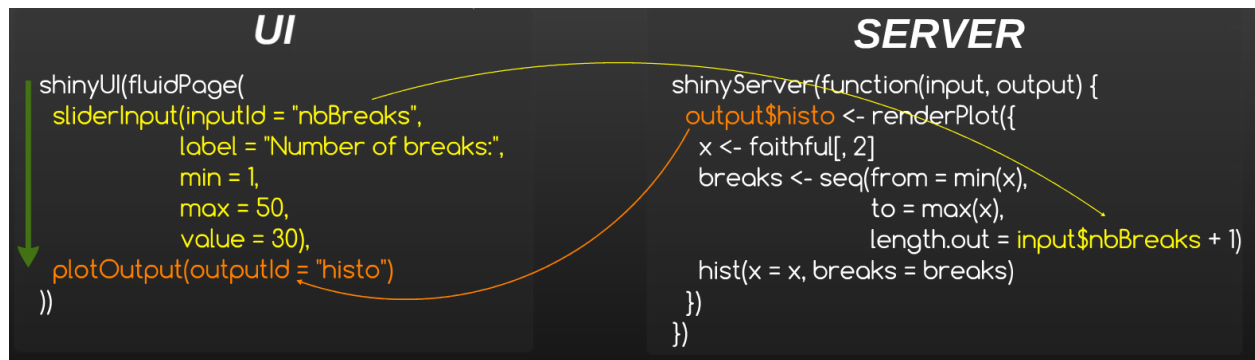
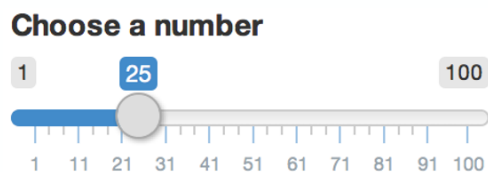


Figure 5: Understand communication

3.4 UI

Deux types d'éléments dans le UI

- `xxInput(inputId = ..., ...)`:
- définit un élément qui permet une action de l'utilisateur
- accessible côté serveur avec son identifiant `input$inputID`



`sliderInput(inputId = "num", label = "Choose a number", ...)`



Figure 6: Understand communication

- `xxOutput(ouputId = ...)`:
- fait référence à un output créé et défini côté serveur
- en général : graphiques et tableaux

3.5 Serveur

Définition des outputs dans le serveur

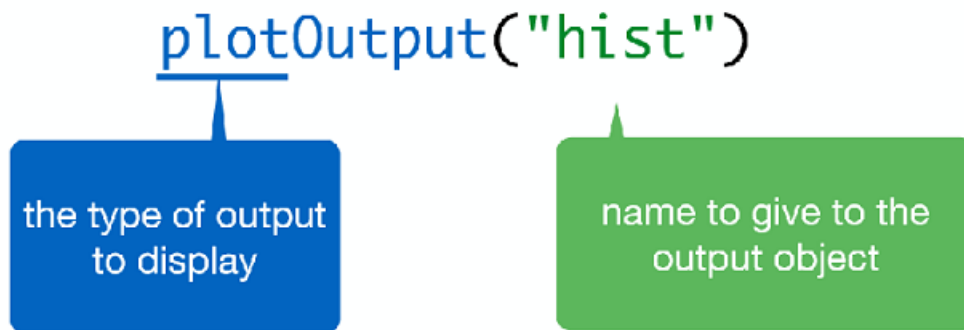


Figure 7: Understand communication

- `renderXX({expr})`:
- calcule et retourne une sortie, dépendante d'inputs, via une expression **R**

```
renderPlot({ hist(rnorm(100)) })
```



Figure 8: Understand communication

3.6 Retour sur le process

C'est plus clair ?

4 Les inputs

4.1 Vue globale

4.2 Valeur numérique

- La fonction

```
numericInput(inputId, label, value, min = NA, max = NA, step = NA)
```

- Exemple:

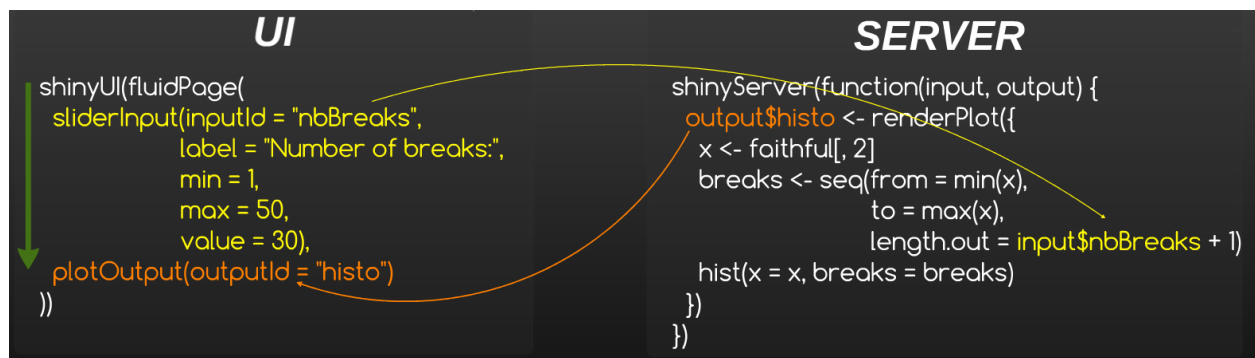


Figure 9: Understand communication

Buttons <input type="button" value="Action"/> <input type="button" value="Submit"/> actionButton() submitButton()	Single checkbox <input checked="" type="checkbox"/> Choice A checkboxInput()	Checkbox group <input checked="" type="checkbox"/> Choice 1 <input type="checkbox"/> Choice 2 <input type="checkbox"/> Choice 3 checkboxGroupInput()	Date input <input type="text" value="2014-01-01"/> dateInput()
Date range <input type="text" value="2014-01-24"/> to <input type="text" value="2014-01-24"/> dateRangeInput()	File input <input type="button" value="Choose File"/> No file chosen fileInput()	accompany other widgets. numericInput()	Password Input <input type="password" value="....."/> passwordInput()
Radio buttons <input checked="" type="radio"/> Choice 1 <input type="radio"/> Choice 2 <input type="radio"/> Choice 3 radioButtons()	Select box <input type="text" value="Choice 1"/> selectInput()	Sliders <input type="range" value="50"/> <input type="range" value="25"/> sliderInput()	Text input <input type="text" value="Enter text..."/> textInput()

Figure 10: Understand communication

```
numericInput(inputId = "idNumeric", label = "Please select a number",
             value = 0, min = 0, max = 100, step = 10)
```

For the server input\$idNumeric will be of class "numeric"
("integer" when the parameter step is an integer value)



Figure 11: numeric

4.3 Chaîne de caractères

- La fonction

```
textInput(inputId, label, value = "")
```

- Exemple:

```
textInput(inputId = "idText", label = "Enter a text", value = "")
```

For the server input\$idText will be of class "character"




Figure 12: text

4.4 Liste de sélection

- La fonction

```
selectInput(inputId, label, choices, selected = NULL, multiple = FALSE,
            selectize = TRUE, width = NULL, size = NULL)
```

- Exemple:


```
selectInput(inputId = "idSelect", label = "Select among the list: ", selected = 3,
           choices = c("First" = 1, "Second" = 2, "Third" = 3))

# For the server input$idSelect is of class "character"
# (vector when the parameter "multiple" is TRUE)
```

Figure 13 shows two examples of the `selectInput` widget. The top example is a single-select dropdown with the label "Select among the list:" and the value "3" selected. The bottom example is a multiple-select dropdown with the label "Select among the list:" and the values "Third" and "Second" selected. To the right of each widget, the "Value" and "Class" are displayed in a box.

Widget	Value	Class
Single-select (3)	[1] "3"	character
Multiple-select (Third, Second)	[1] "3" "2"	character

Figure 13: us

4.5 Checkbox

- La fonction

```
checkboxInput(inputId, label, value = FALSE)
```

- Exemple:

```
checkboxInput(inputId = "idCheck1", label = "Check ?")

# For the server input$idCheck1 is of class "logical"
```

Figure 14 shows a checkbox widget with the label "checkboxInput" and "Check ?". The checkbox is checked. To the right, the "Value" is displayed as [1] TRUE and the "Class" is displayed as logical.

Widget	Value	Class
checkboxInput (checked)	[1] TRUE	logical

Figure 14: sc

4.6 Checkboxes multiple

- La fonction

```
checkboxGroupInput(inputId, label, choices, selected = NULL, inline = FALSE)
```

- Exemple:

```
checkboxGroupInput(inputId = "idCheckGroup", label = "Please select", selected = 3,  
  choices = c("First" = 1, "Second" = 2, "Third" = 3))  
  
# For the server input$idCheckGroup is a "character" vector
```

Please select

☐ First

☒ Second

☒ Third

Value: [1] "2" "3"

Class: character

Figure 15: mc

4.7 Radio boutons

- La fonction

```
radioButtons(inputId, label, choices, selected = NULL, inline = FALSE)
```

- Exemple:

```
radioButtons(inputId = "idRadio", label = "Select one", selected = 3,  
  choices = c("First" = 1, "Second" = 2, "Third" = 3))  
  
# For the server input$idRadio is a "character"
```

Select one

☐ First

☐ Second

☒ Third

Value: [1] "3"

Class: character

Figure 16: rb

4.8 Date

- La fonction

```
dateInput(inputId, label, value = NULL, min = NULL, max = NULL, format = "yyyy-mm-dd",
          startview = "month", weekstart = 0, language = "en")
```

- Exemple:

```
dateInput(inputId = "idDate", label = "Please enter a date", value = "12/08/2015",
          format = "dd/mm/yyyy", startview = "month", weekstart = 0, language = "fr")
```

For the server input\$idDate is a "Date"

Figure 17 shows a date input field with the label "Please enter a date" and the value "07/12/2015". To the right of the input field, the "Value:" is displayed as "[1] \"2015-12-07\"" and the "Class:" is displayed as "Date".

Figure 17: d

4.9 Période

- La fonction

```
dateRangeInput(inputId, label, start = NULL, end = NULL, min = NULL, max = NULL,
                format = "yyyy-mm-dd", startview = "month", weekstart = 0,
                language = "en", separator = " to ")
```

- Exemple:

```
dateRangeInput(inputId = "idDateRange", label = "Please Select a date range",
               start = "2015-01-01", end = "2015-08-12", format = "yyyy-mm-dd",
               language = "en", separator = " to ")
```

For the server input\$idDateRange is a vector of class "Date" with two elements

Figure 18 shows a date range input field with the label "Please Select a date range" and the value "2015-01-01 to 2015-08-12". To the right of the input field, the "Value:" is displayed as "[1] \"2015-01-01\" \"2015-08-12\"" and the "Class:" is displayed as "Date".

Figure 18: dr

4.10 Slider numérique : valeur unique

- La fonction

```
sliderInput(inputId, label, min, max, value, step = NULL, round = FALSE,  
            format = NULL, locale = NULL, ticks = TRUE, animate = FALSE,  
            width = NULL, sep = ",", pre = NULL, post = NULL)
```

- Exemple:

```
sliderInput(inputId = "idSlider1", label = "Select a number", min = 0, max = 10,  
            value = 5, step = 1)
```

```
# For the server input$idSlider1 is a "numeric"  
# (integer when the parameter "step" is an integer too)
```



Figure 19: sl

4.11 Slider numérique : range

- La fonction

```
sliderInput(inputId, label, min, max, value, step = NULL, round = FALSE,  
            format = NULL, locale = NULL, ticks = TRUE, animate = FALSE,  
            width = NULL, sep = ",", pre = NULL, post = NULL)
```

- Exemple:

```
sliderInput(inputId = "idSlider2", label = "Select a number", min = 0, max = 10,  
            value = c(2,7), step = 1)
```

```
# For the server input$idSlider2 is a "numeric" vector  
# (integer when the parameter "step" is an integer too)
```

4.12 Importer un fichier

- La fonction



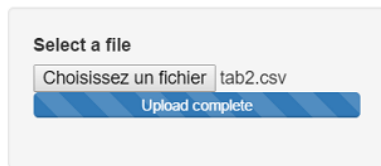
Figure 20: msl

```
fileInput(inputId, label, multiple = FALSE, accept = NULL)
```

- Exemple:

```
fileInput(inputId = "idFile", label = "Select a file")

# For the server input$idFile is a "data.frame" with four "character" columns
# (name, size, type and datapath) and one row
```



Value:

	name	size	type	datapath
1	tab2.csv	40	application/vnd.ms-excel	C:\Users\Benoit\AppData

Figure 21: up

4.13 Action Bouton

- La fonction

```
actionButton(inputId, label, icon = NULL, ...)
```

- Exemple:

```
actionButton(inputId = "idActionButton", label = "Click !",
             icon = icon("hand-spock-o"))

# For the server input$idActionButton is an "integer"
```

4.14 Aller plus loin : construire son propre input

Avec un peu de compétences en HTML/CSS/JavaScript, il est également possible de construire des inputs personnalisés

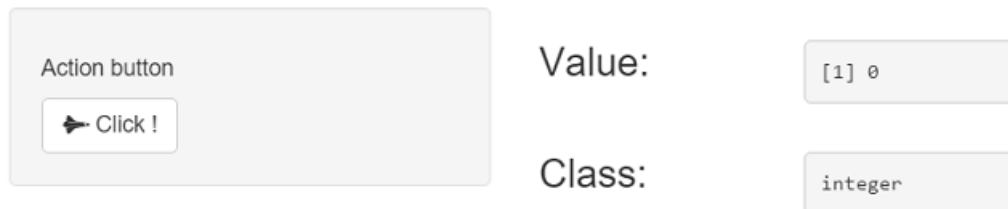


Figure 22: ab

Un tutoriel est disponible : <http://shiny.rstudio.com/articles/building-inputs.html>

Ainsi que deux applications d'exemples :

- <http://shiny.rstudio.com/gallery/custom-input-control.html>
- <http://shiny.rstudio.com/gallery/custom-input-bindings.html>

5 Outputs

5.1 Vue globale

server fonction	ui fonction	type de sortie
<code>renderDataTable()</code>	<code>dataTableOutput()</code>	une table interactive
<code>renderImage()</code>	<code>imageOutput()</code>	une image sauvegardée
<code>renderPlot()</code>	<code>plotOutput</code>	un graphique R
<code>renderPrint()</code>	<code>verbatimTextOutput()</code>	affichage type console R
<code>renderTable()</code>	<code>tableOutput()</code>	une table statique
<code>renderText()</code>	<code>textOutput()</code>	une chaîne de caractère
<code>renderUI()</code>	<code>uiOutput()</code>	un élément de type UI

Figure 23: ab

5.2 Les bonnes règles de construction

- assigner l'output à afficher dans la liste **output**, avec un nom permettant l'identification côté **UI**
- utiliser une fonction **renderXX({expr})**
- **la dernière expression doit correspondre au type d'objet retourné**
- accéder aux inputs, et amener la réactivité, en utilisant la liste **input** et l'identifiant : **input\$inputId**

```
#ui.R
selectInput("lettre", "Lettres:", LETTERS[1:3])
verbatimTextOutput(outputId = "selection")
#server.R
output$selection <- renderPrint({input$lettre})
```

5.3 Print

- ui.r:

```
verbatimTextOutput(outputId = "texte")
```

- server.r:

```
output$texte <- renderPrint({  
  c("Hello shiny !")  
})
```

```
[1] "Hello shiny !"
```

Figure 24: op

5.4 Text

- ui.r:

```
textOutput(outputId = "texte")
```

- server.r:

```
output$texte <- renderText({  
  c("Hello shiny !")  
})
```

Hello shiny !

Figure 25: op2

5.4.1 Plot

- ui.r:

```
plotOutput("myplot")
```

- server.r:

```
output$myplot <- renderPlot({  
  hist(iris$Sepal.Length)  
})
```

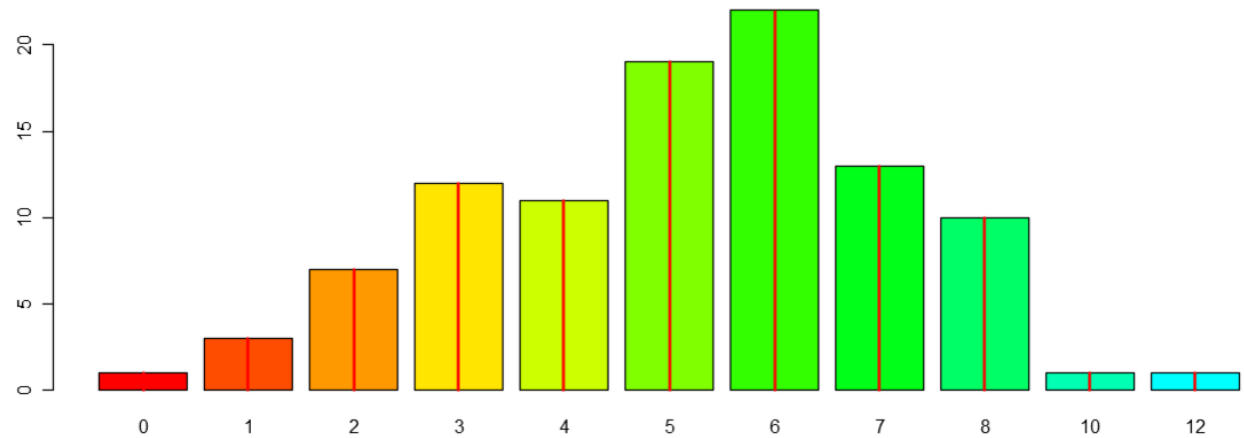


Figure 26: op3

5.5 Table

- ui.r:

```
tableOutput(outputId = "table")
```

- server.r:

```
output$table <- renderTable({iris})
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.10	3.50	1.40	0.20	setosa
2	4.90	3.00	1.40	0.20	setosa
3	4.70	3.20	1.30	0.20	setosa
4	4.60	3.10	1.50	0.20	setosa
5	5.00	3.60	1.40	0.20	setosa

Figure 27: op4

5.6 DataTable

- ui.r:


```
dataTableOutput(outputId = "dataTable")
```

- server.r:

```
output$dataTable <- renderDataTable({
  iris
})
```

Showing 1 to 5 of 5 entries

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa

Showing 1 to 5 of 5 entries

Figure 28: op5

5.7 Définir des éléments de l'UI côté SERVER | Définition

Dans certains cas, nous souhaitons définir des inputs ou des structures côté server

L'exemple typique étant de créer un input dépendant d'un fichier utilisateur, comme lister les colonnes présentes

Cela est possible avec les fonctions `uiOutput` et `renderUI`

5.8 Définir des éléments de l'UI côté SERVER | Exemple simple

- ui.r:

```
uiOutput(outputId = "columns")
```

- server.r:

```
output$columns <- renderUI({
  selectInput(inputId = "sel_col", label = "Column", choices = colnames(data))
})
```

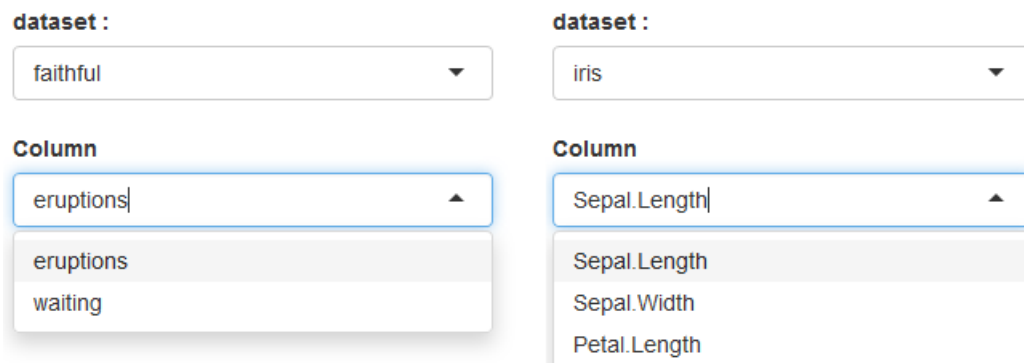


Figure 29: op5

5.9 Définir des éléments de l'UI côté SERVER | Exemple plus complexe

- On peut également renvoyer un élément plus complexe de l'UI, par exemple :
 - tout en layout
 - ou une `fluidRow`
- `ui.r`:

```
uiOutput(outputId = "fluidRow_ui")
```

- `server.r`:

```
output$fluidRow_ui <- renderUI(
  fluidRow(
    column(width = 3, h3("Value:")),
    column(width = 3, h3(verbatimTextOutput(outputId = "slinderIn_value")))
  )
)
```

5.10 Aller plus loin : construire son propre output

Avec un peu de compétences en HTML/CSS/JavaScript, il est également possible de construire des outputs personnalisés

Un tutoriel est disponible : <http://shiny.rstudio.com/articles/building-outputs.html>

On peut donc par exemple ajouter comme output un graphique construit avec la librairie d3.js. Un exemple est disponible dans le dossier `shinyApps/build_output`.

6 Structure d'une application

6.1 Un seul fichier

- enregistré sous le nom **app.R**
- se terminant par la commande `shinyApp()`

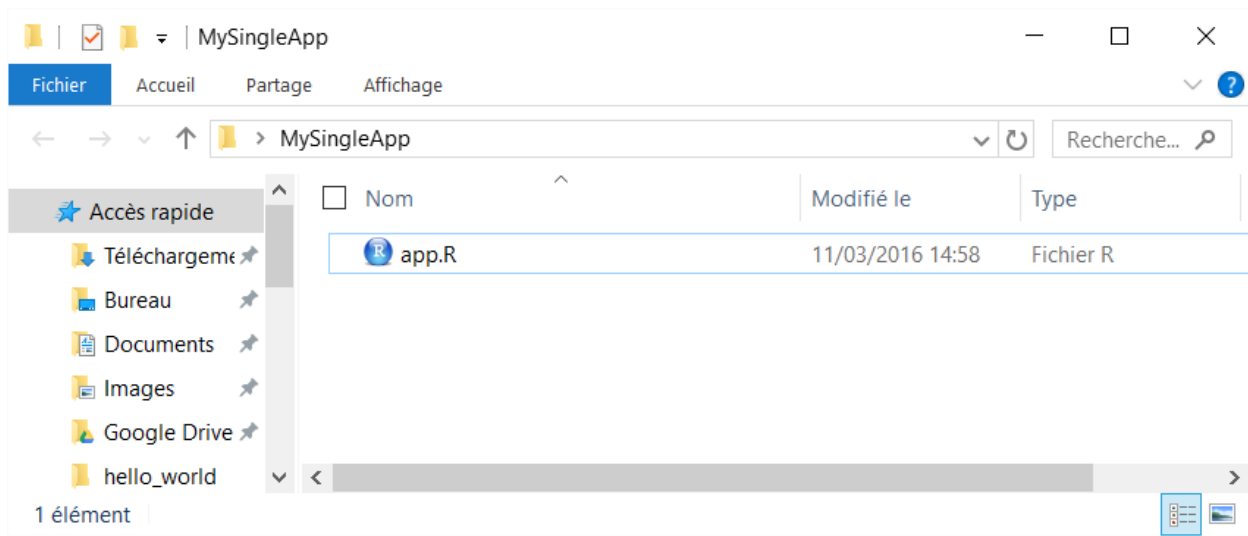


Figure 30: Understand communication

- pour les applications légères

```
library(shiny)
ui <- fluidPage(
  sliderInput(inputId = "num", label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$num))
  })
}
shinyApp(ui = ui, server = server)
```

6.2 Deux fichiers

- côté interface utilisateur dans le script **ui.R**
- côté serveur dans le script **server.R**

ui.R

```
library(shiny)
fluidPage(
  sliderInput(inputId = "num", label = "Choose a number",
    value = 25, min = 1, max = 100),
  plotOutput("hist")
)
```

server.R

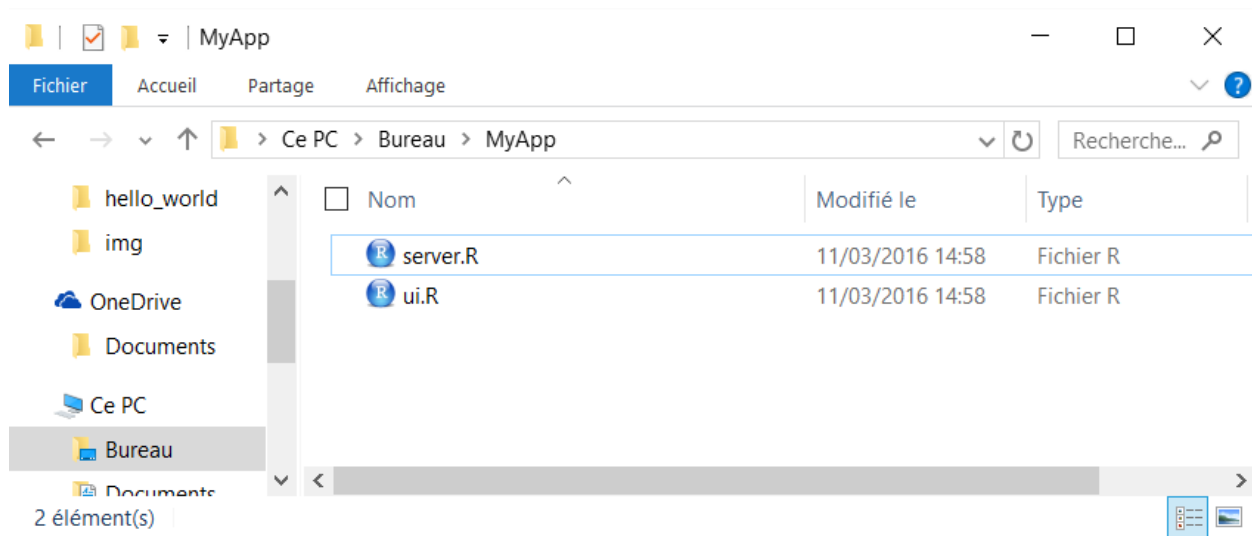


Figure 31: Understand communication

```
library(shiny)
function(input, output) {
  output$hist <- renderPlot({hist(rnorm(input$num))})
}
```

6.3 UI en HTML

Même si en général on code le **UI** dans un **script R**, il est également possible de le coder entièrement dans un fichier **.html**. Plus d'informations ici

- côté interface utilisateur, un fichier **index.html** dans le répertoire **www**
- côté serveur dans le script **server.R**

6.4 Données/fichiers complémentaires

- le code **R** tourne au niveau des scripts **R**, et peut donc accéder de façon relative à tous les objets présents dans le dossier de l'application
- l'application web, comme de convention, accède à tous les éléments présents dans le dossier **www**

6.5 Partage ui <-> server

Le server et l'ui communiquent uniquement par le biais des **inputs** et des **outputs**

- Nous pouvons ajouter un script nommé **global.R** pour partager des éléments (variables, packages, ...) entre la partie **UI** et la partie **SERVER**
- Tout ce qui est présent dans le **global.R** est visible à la fois dans le **ui.R** et dans le **server.R**
- Le script **global.R** est chargé uniquement une seule fois au lancement de l'application
- Dans le cas d'une utilisation avec un **shiny-server**, les objets globaux sont également partagés entre les utilisateurs

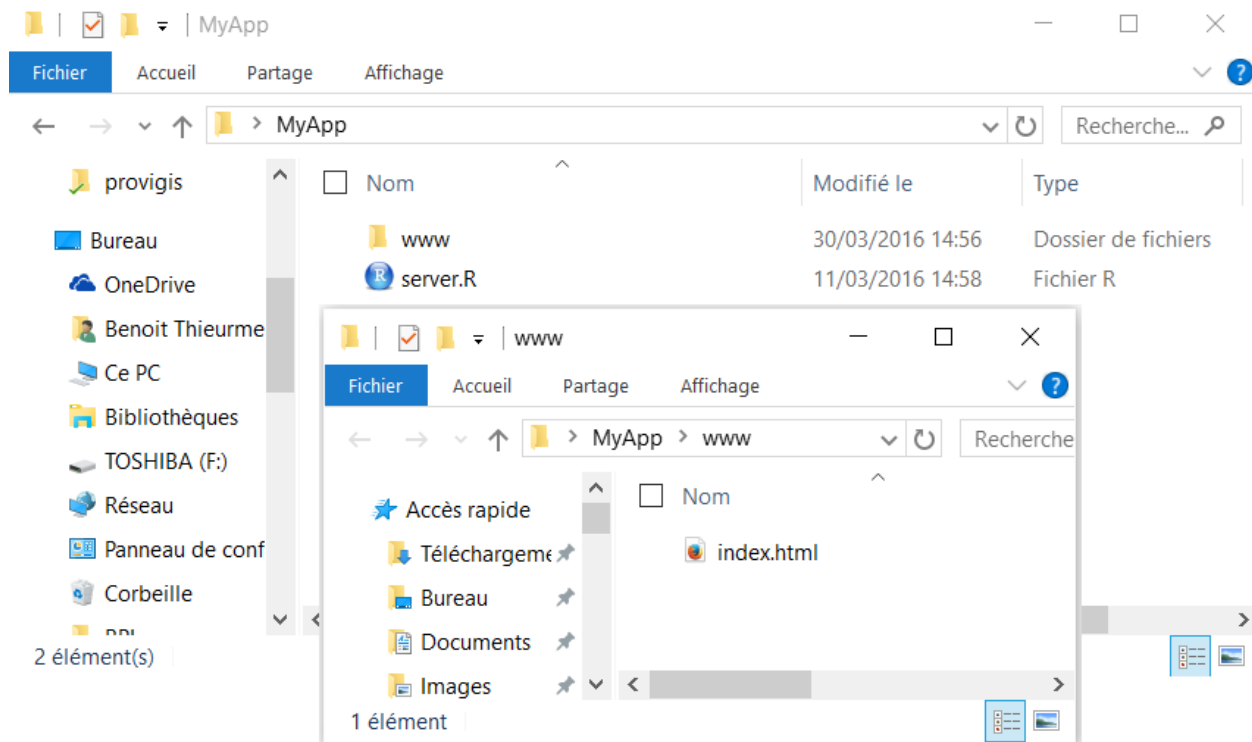


Figure 32:

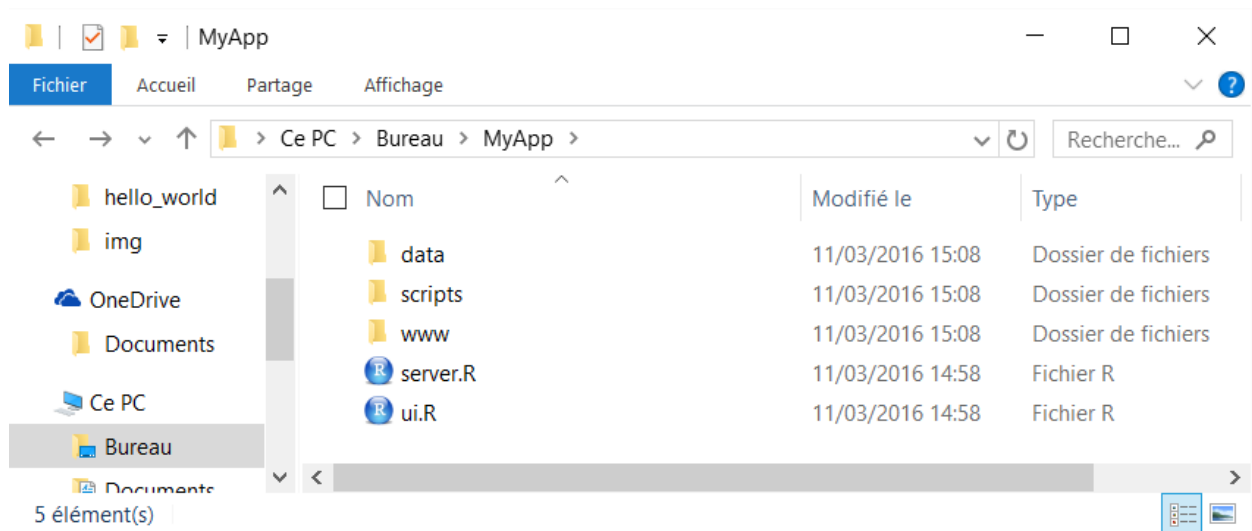


Figure 33: Understand communication

7 Structurer sa page

7.1 sidebarLayout

Le template basique `sidebarLayout` divise la page en deux colonnes et doit contenir :

- `sidebarPanel`, à gauche, en général pour les inputs
- `mainPanel`, à droite, en général pour les outputs

```
shinyUI(fluidPage(  
  titlePanel("Old Faithful Geyser Data"), # title  
  sidebarLayout(  
    sidebarPanel("SIDEBAR"),  
    mainPanel("MAINPANEL")  
  )  
))
```

My first app

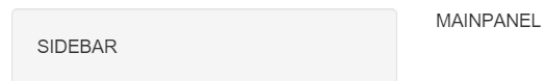


Figure 34: sdb

7.2 wellPanel

Comme avec le `sidebarPanel` précédent, on peut griser un ensemble d'éléments en utilisant un `wellPanel` :

```
shinyUI(fluidPage(  
  titlePanel("Old Faithful Geyser Data"), # title  
  wellPanel(  
    sliderInput("num", "Choose a number", value = 25, min = 1, max = 100),  
    textInput("title", value = "Histogram", label = "Write a title")  
  ),  
  plotOutput("hist")  
))
```

7.3 navbarPage

Utiliser une barre de navigation et des onglets avec `navbarPage` et `tabPanel`:

Without wellPanel

Choose a number

1 25 100

1 11 21 31 41 51 61 71 81 91 100

Write a title

Histogram

With wellPanel

Choose a number

1 25 100

1 11 21 31 41 51 61 71 81 91 100

Write a title

Histogram

Figure 35: sdb

```
shinyUI(
  navbarPage(
    title = "My first app",
    tabPanel(title = "Summary",
      "Here is the summary"),
    tabPanel(title = "Plot",
      "some charts"),
    tabPanel(title = "Table",
      "some tables")
  )
)
```

Nous pouvons rajouter un second niveau de navigation avec un `navbarMenu` :

```
shinyUI(
  navbarPage(
    title = "My first app",
    tabPanel(title = "Summary",
      "Here is the summary"),
    tabPanel(title = "Plot",
      "some charts"),
    navbarMenu("Table",
      tabPanel("Table 1"),
      tabPanel("Table 2")
    )
  )
)
```

7.4 tabsetPanel

Plus généralement, nous pouvons créer des onglets à n'importe quel endroit en utilisant `tabsetPanel` & `tabPanel`:

```
shinyUI(fluidPage(
  titlePanel("Old Faithful Geyser Data"), # title
  sidebarLayout(
```



Figure 36: nav

```

sidebarPanel("SIDEBAR"),
mainPanel(
  tabsetPanel(
    tabPanel("Plot", plotOutput("plot")),
    tabPanel("Summary", verbatimTextOutput("summary")),
    tabPanel("Table", tableOutput("table"))
  )
)
))

```

My first app

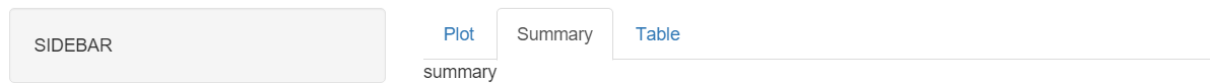


Figure 37: pan

7.5 navlistPanel

Une alternative au `tabsetPanel`, pour une disposition verticale plutôt qu'horizontale : `navlistPanel`

```

shinyUI(fluidPage(
  navlistPanel(
    tabPanel("Plot", plotOutput("plot")),
    tabPanel("Summary", verbatimTextOutput("summary")),
    tabPanel("Table", tableOutput("table"))
  )
))

```

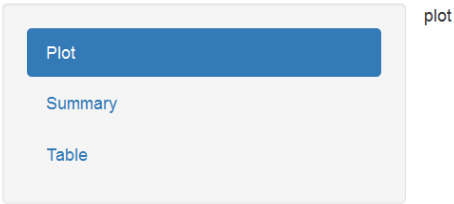



Figure 38: pan

7.6 Grid Layout

Créer sa propre organisation avec `fluidRow()` et `column()`

- chaque ligne peut être divisée en 12 colonnes
- le dimensionnement final de la page est automatique en fonction des éléments dans les lignes / colonnes

```
tabPanel(title = "Summary",
  # A fluid row can contain from 0 to 12 columns
  fluidRow(
    # A column is defined necessarily
    # with its argument "width"
    column(width = 4, "column 1"),
    column(width = 4, "column 2"),
    column(width = 4, "column 3"),
  ))
```

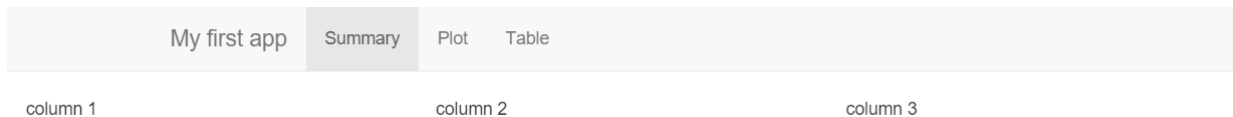


Figure 39: grid

7.7 Inclure du HTML

De nombreuses de balises **html** sont disponibles avec les fonctions `tags` :

```
names(shiny::tags)
```

```
##   [1] "a"           "abbr"        "address"     "area"        "article"
##   [6] "aside"       "audio"       "b"           "base"        "bdi"
##  [11] "bdo"        "blockquote"  "body"       "br"          "button"
##  [16] "canvas"     "caption"     "cite"       "code"        "col"
##  [21] "colgroup"   "command"    "data"       "datalist"    "dd"
```

## [26]	"del"	"details"	"dfn"	"div"	"dl"
## [31]	"dt"	"em"	"embed"	"eventsource"	"fieldset"
## [36]	"figcaption"	"figure"	"footer"	"form"	"h1"
## [41]	"h2"	"h3"	"h4"	"h5"	"h6"
## [46]	"head"	"header"	"hgroup"	"hr"	"html"
## [51]	"i"	"iframe"	"img"	"input"	"ins"
## [56]	"kbd"	"keygen"	"label"	"legend"	"li"
## [61]	"link"	"mark"	"map"	"menu"	"meta"
## [66]	"meter"	"nav"	"noscript"	"object"	"ol"
## [71]	"optgroup"	"option"	"output"	"p"	"param"
## [76]	"pre"	"progress"	"q"	"ruby"	"rp"
## [81]	"rt"	"s"	"samp"	"script"	"section"
## [86]	"select"	"small"	"source"	"span"	"strong"
## [91]	"style"	"sub"	"summary"	"sup"	"table"
## [96]	"tbody"	"td"	"textarea"	"tfoot"	"th"
## [101]	"thead"	"time"	"title"	"tr"	"track"
## [106]	"u"	"ul"	"var"	"video"	"wbr"



Figure 40: grid

C'est également possible de passer du code **HTML** directement en utilisant la fonction du même nom :

```
fluidPage(
  HTML("<h1>My Shiny App</h1>")
)
```

7.8 shinydashboard

Le package shinydashboard propose d'autres fonctions pour créer des tableaux de bords :

<https://rstudio.github.io/shinydashboard/>

7.9 Combiner les structures

Toutes les structures peuvent s'utiliser en même temps !

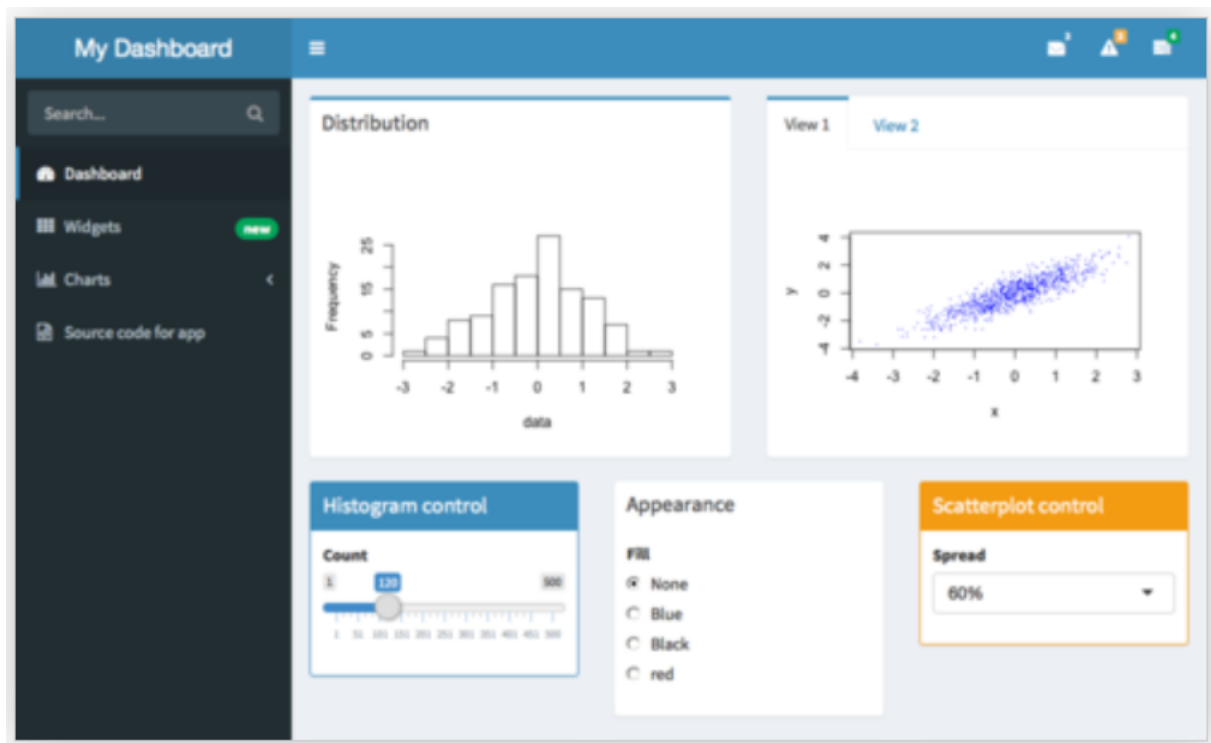


Figure 41: grid

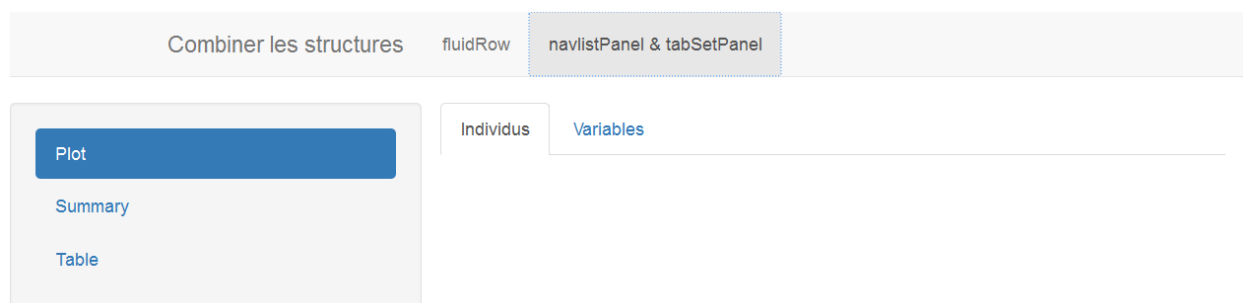


Figure 42: grid

8 Customisation avec du CSS

Shiny utilise Bootstrap pour la partie **CSS**.

Comme dans du développement web “classique”, nous pouvons modifier le **CSS** de trois façons :

- en faisant un lien vers un fichier .css externe, en ajoutant des feuilles de style dans le répertoire **www**
- en ajoutant du **CSS** dans le header **HTML**
- en écrivant individuellement du CSS aux éléments.

Il y a une notion d'ordre et de priorité sur ces trois informations : le **CSS** “individuel” l’emporte sur le **CSS** du header, qui l’emporte sur le **CSS** externe

On peut aussi utiliser le package shinythemes

8.1 Avec un .css externe

On peut par exemple aller prendre un thème sur bootswatch.

- Deux façons pour le renseigner :
- argument **theme** dans **fluidPage**
- ou avec un tags html : **tags\$head** et **tags\$link**

```
library(shiny)
ui <- fluidPage(theme = "mytheme.css",
  # ou avec un tags
  tags$head(
    tags$link(rel = "stylesheet", type = "text/css", href = "mytheme.css")
  ),
  # reste de l'application
)
```

8.2 Ajout de css dans le header

- Le **CSS** inclut dans le header sera prioritaire au **CSS** externe
- inclusion avec les tags html : **tags\$head** et **tags\$style**

```
library(shiny)
tags$head(
  tags$style(HTML("h1 { color: #48ca3b;}"))
),
# reste de l'application
)
```

8.3 CSS sur un élément

Pour finir, on peut également passer directement du **CSS** aux éléments **HTML** :

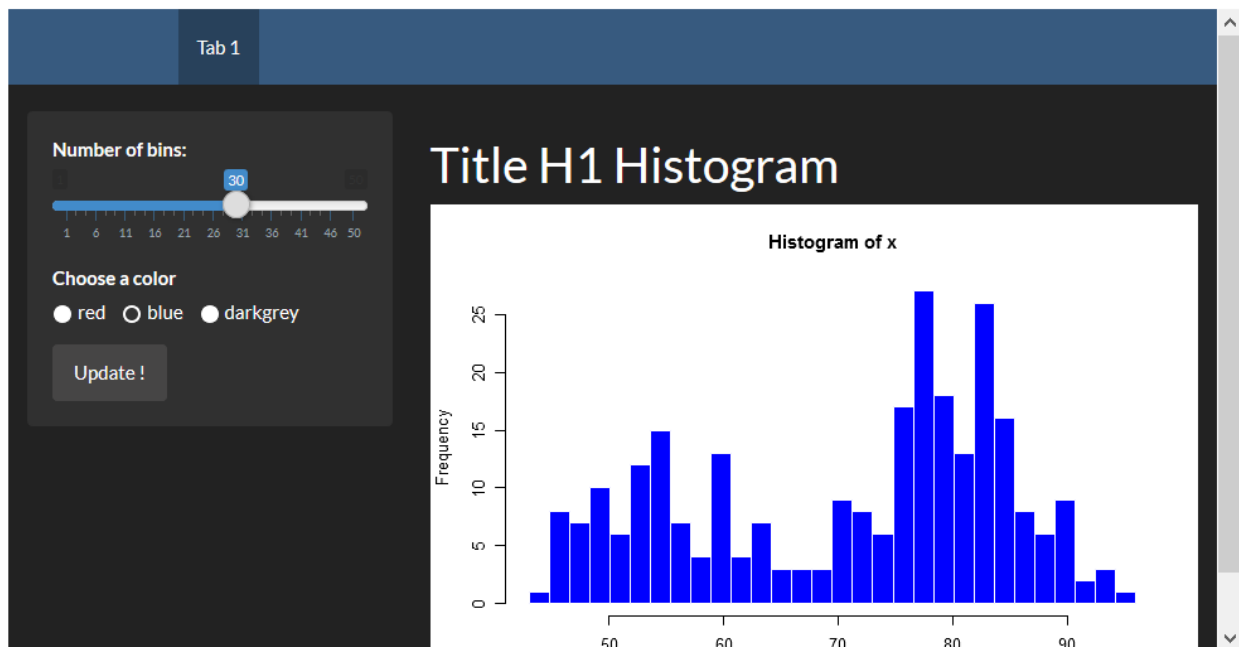


Figure 43: css1

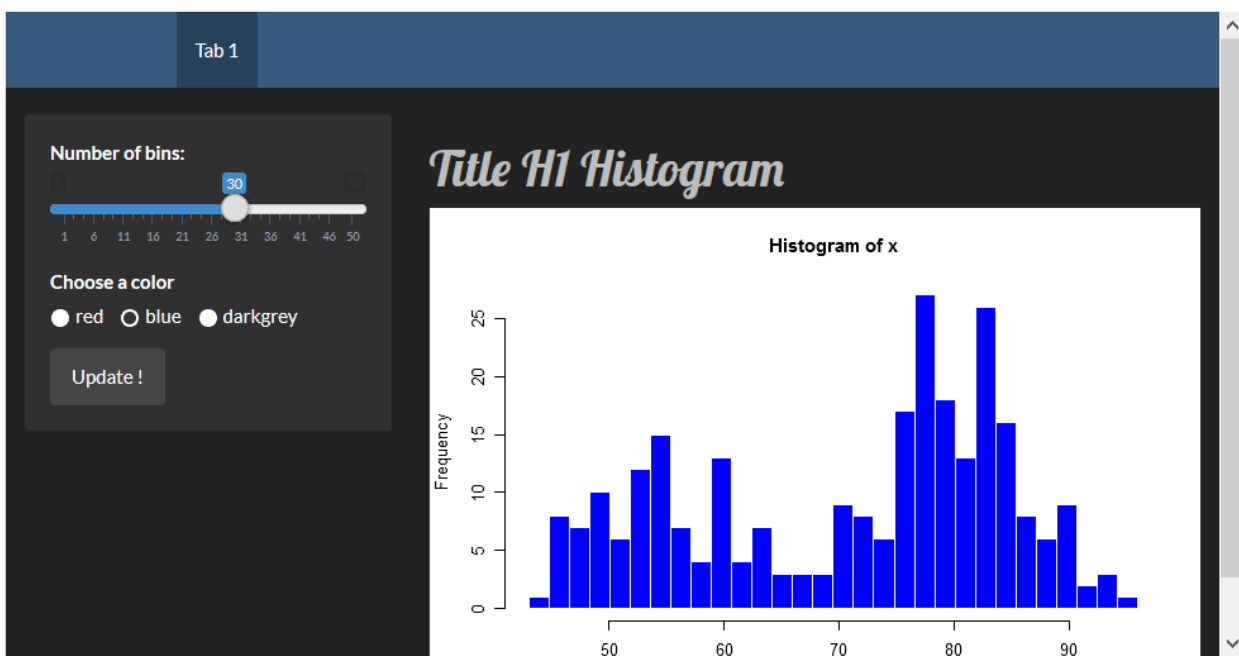


Figure 44: css2

```
library(shiny)
h1("Mon titre", style = "color: #48ca3b;")
# reste de l'application
)
```

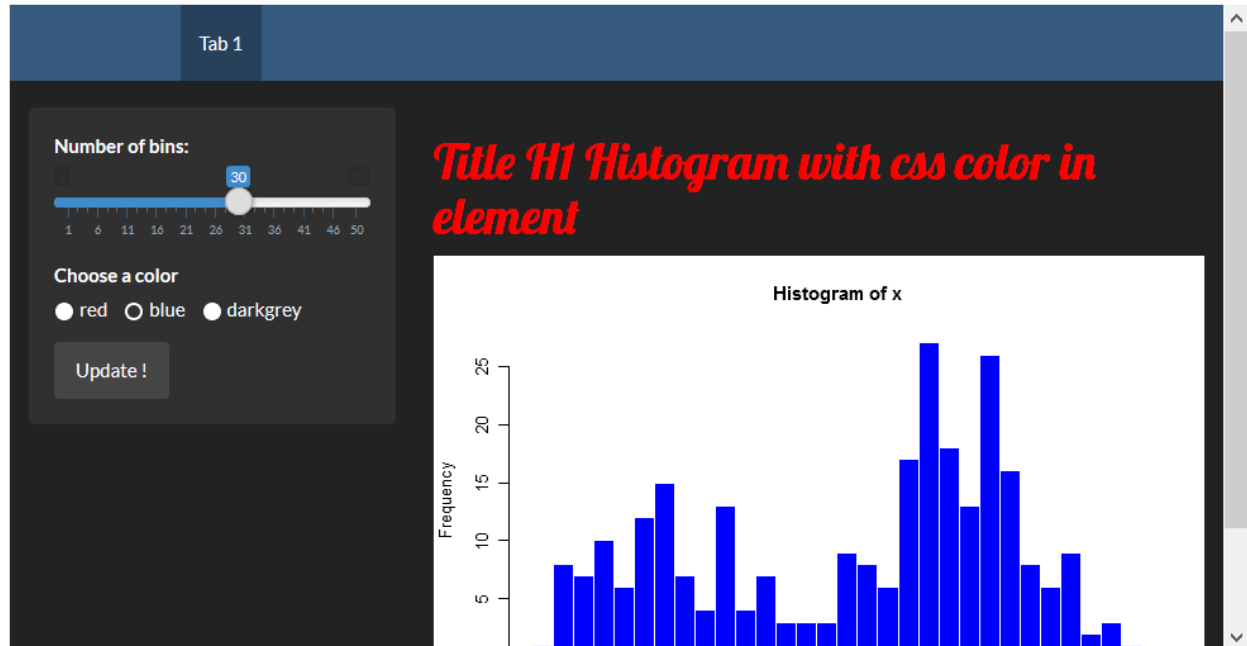


Figure 45: css3

9 Graphiques interactifs

Avec notamment l'arrivée du package `htmlwidgets`, de plus en plus de fonctionnalités de librairies javascript sont accessibles sous **R** :

- `dygraphs` (time series)
- `DT` (interactive tables)
- `Leaflet` (maps)
- `d3heatmap`
- `threejs` (3d scatter & globe)
- `rAmCharts`
- `visNetwork`
- ...

Plus généralement, jeter un oeil sur la gallerie suivante!

9.1 Utilisation dans shiny

Tous ces packages sont utilisables simplement dans **shiny**. En effet, ils contiennent les deux fonctions nécessaires :

- **renderXX**
- **xxOutput**

Par exemple avec le package `dygraphs` :

```
# UI
output$dygraph <- renderDygraph({
  dygraph(predicted(), main = "Predicted Deaths/Month")
})
# SERVER
dygraphOutput("dygraph")
```

dygraphs

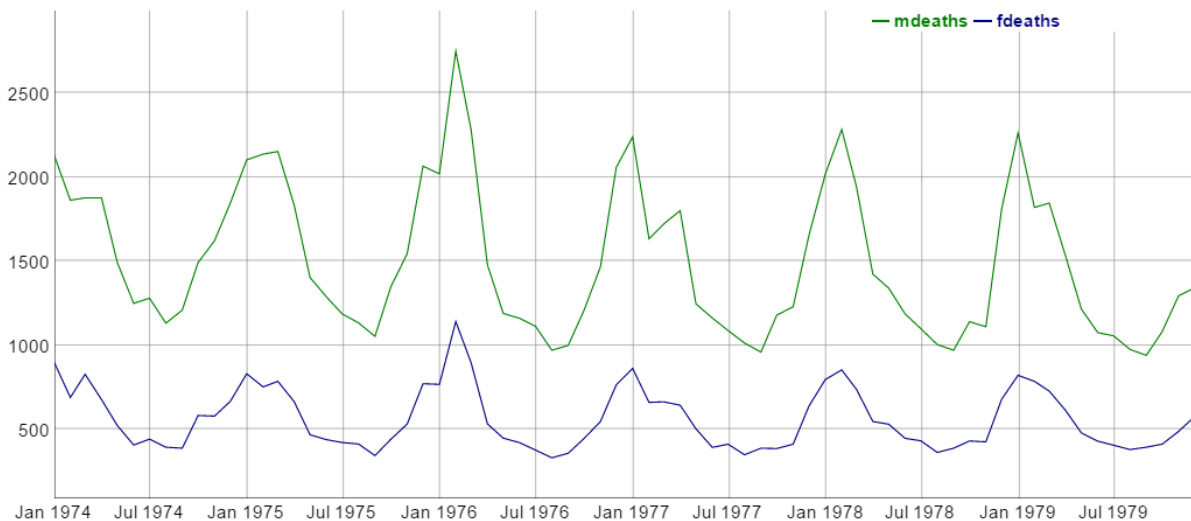


Figure 46: dyg

10 Isolation

10.1 Définition

Par défaut, les outputs et les expressions réactives se mettent à jour automatiquement quand un des inputs présents dans le code change de valeur. Dans certains cas, on aimerait pouvoir contrôler un peu cela.

Par exemple, en utilisant un bouton de validation (**actionButton**) des inputs pour déclencher le calcul des sorties.

- un input peut être isolé comme cela `isolate(input$id)`
- une expression avec la notation suivante `isolate({expr})` et l'utilisation de `{}`

10.2 Exemple 1

- **ui.r**: Trois inputs : **color** et **bins** pour l'histogramme, et un **actionButton** :

leaflet

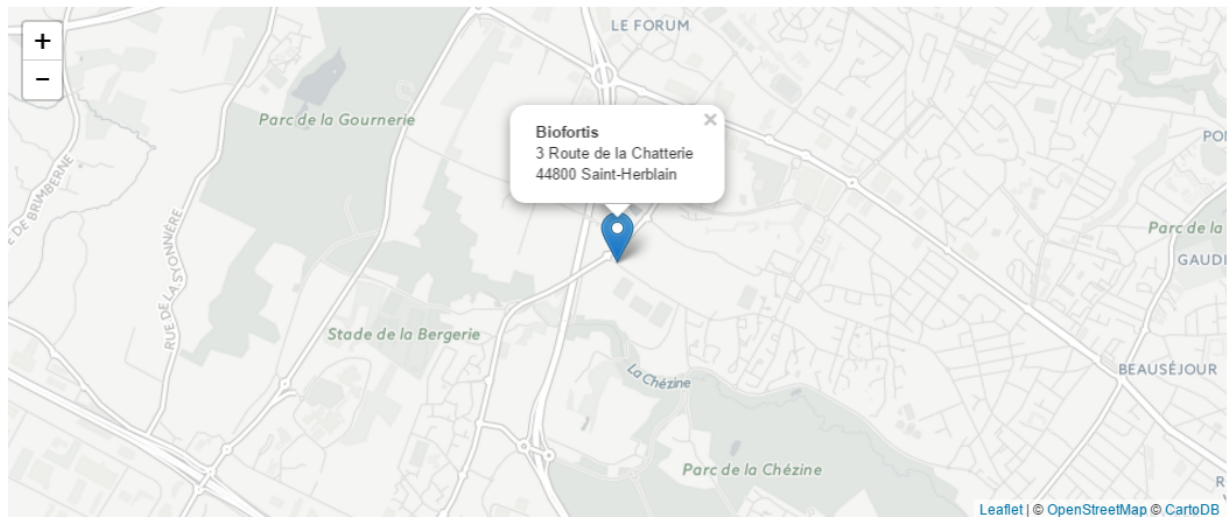


Figure 47: leaf

rAmCharts

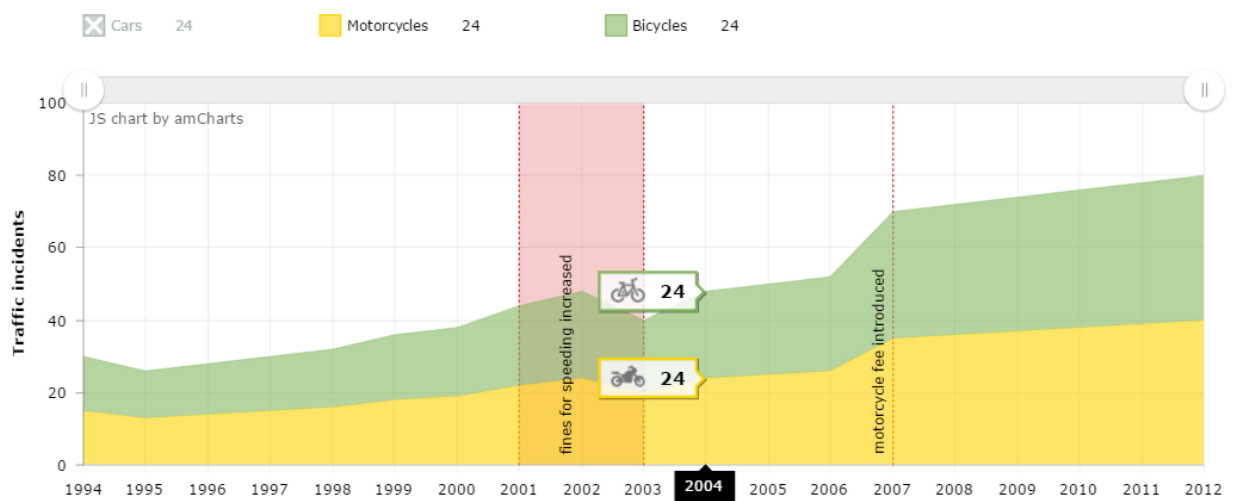


Figure 48: ram

visNetwork

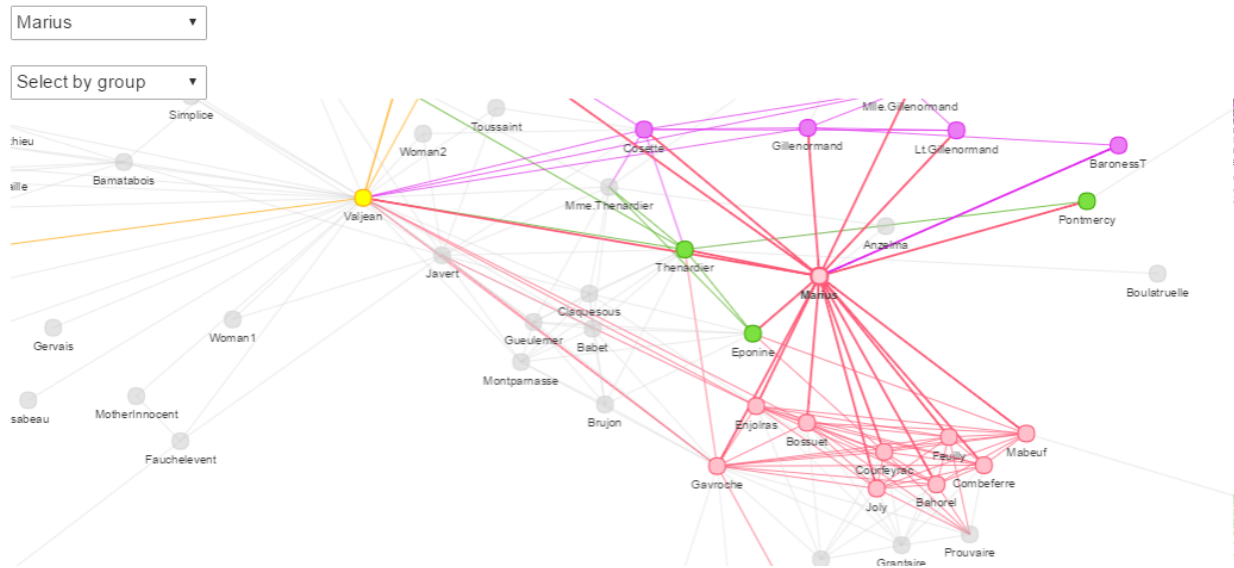
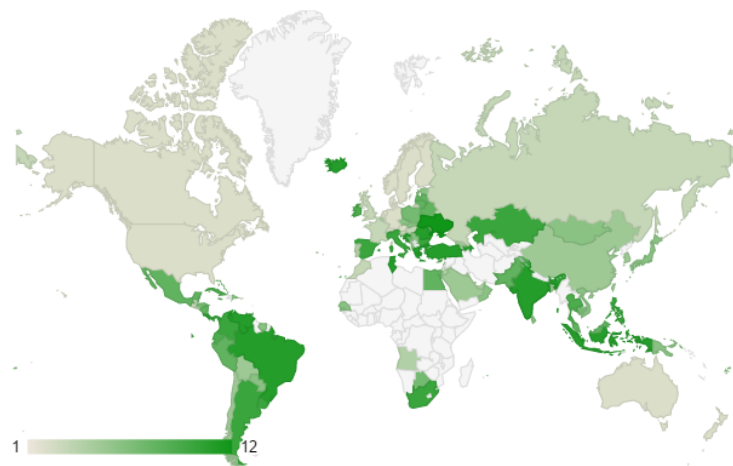


Figure 49: vis

googleVis Example



Data from:

https://en.wikipedia.org/wiki/List_of_countries_by_credit_rating

Figure 50: ggvis

```
shinyUI(fluidPage(
  titlePanel("Isolation"),
  sidebarLayout(
    sidebarPanel(
      radioButtons(inputId = "col", label = "Choose a color", inline = TRUE,
        choices = c("red", "blue", "darkgrey")),
      sliderInput("bins", "Number of bins:", min = 1, max = 50, value = 30),
      actionButton("go_graph", "Update !")
    ),
    mainPanel(plotOutput("distPlot"))
  )
))
```

- **server.r:**

On isole tout le code sauf l'**actionButton** :

```
shinyServer(function(input, output) {
  output$distPlot <- renderPlot({
    input$go_graph
    isolate({
      inputColor <- input$color
      x <- faithful[, 2]
      bins <- seq(min(x), max(x), length.out = input$bins + 1)
      hist(x, breaks = bins, col = inputColor, border = 'white')
    })
  })
})
```

L'histogramme sera donc mis-à-jour quand l'utilisateur cliquera sur le bouton.

Isolation

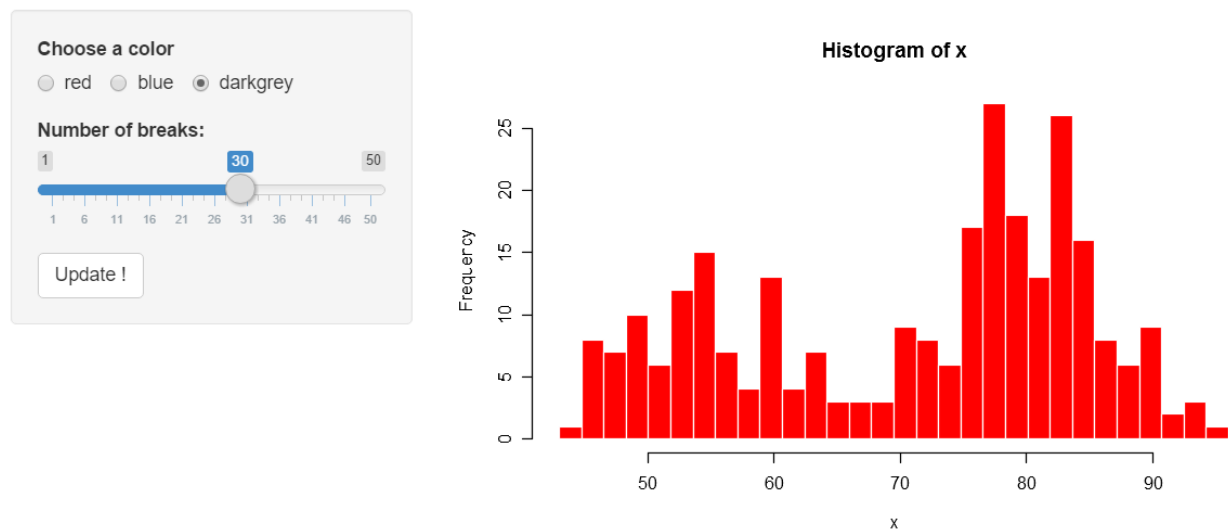


Figure 51: iso

10.3 Exemple 2

- **server.R**:

```
output$distPlot <- renderPlot({
  input$go_graph
  inputColor <- input$color
  isolate({
    x <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = bins, col = inputColor, border = 'white')
  })
})
```

Même résultat en isolant seulement le troisième et dernier input `input$bins`

```
input$go_graph
x <- faithful[, 2]
bins <- seq(min(x), max(x), length.out = isolate(input$bins) + 1)
hist(x, breaks = bins, col = input$color, border = 'white')
```

L'histogramme sera donc mis-à-jour quand l'utilisateur cliquera sur le bouton ou quand la couleur changera.

11 Expressions réactives

Les expressions réactives sont très utiles quand on souhaite utiliser le même résultat/objet dans plusieurs outputs, en ne faisant le calcul qu'une fois.

Il suffit pour cela d'utiliser la fonction **reactive** dans le **server.R**

Par exemple, nous voulons afficher deux graphiques à la suite d'une ACP:

- La projection des individus
- La projection des variables

11.1 Exemple sans une expression réactive

- **server.R**: le calcul est réalisé deux fois...

```
require(FactoMineR) ; data("decathlon")

output$graph_pca_ind <- renderPlot({
  res_pca <- PCA(decathlon[, input$variables], graph = FALSE)
  plot.PCA(res_pca, choix = "ind", axes = c(1,2))
})

output$graph_pca_var <- renderPlot({
  res_pca <- PCA(decathlon[, input$variables], graph = FALSE)
  plot.PCA(res_pca, choix = "var", axes = c(1,2))
})
```

11.2 Exemple avec une expression réactive

- **server.R** : Le calcul est maintenant effectué qu’une seule fois !

```
require(FactoMineR) ; data("decathlon")

res_pca <- reactive({
  PCA(decathlon[,input$variables], graph = FALSE)
})

output$graph_pca_ind <- renderPlot({
  plot.PCA(res_pca(), choix = "ind", axes = c(1,2))
})

output$graph_pca_var <- renderPlot({
  plot.PCA(res_pca(), choix = "var", axes = c(1,2))
})
```

11.3 Note

- Une expression réactive va nous faire gagner du temps et de la mémoire
- **Utiliser des expressions réactives seulement quand cela dépend d’inputs** (pour d’autres variables : <http://shiny.rstudio.com/articles/scoping.html>)
- **Comme un output** : mis-à-jour chaque fois qu’un input présent dans le code change
- **Comme un input** dans un *renderXX* : l’output est mis-à-jour quand l’expression réactive change
- On récupère sa valeur comme un appel à une fonction, avec des “()”.

11.4 reactiveValues

Il existe une alternative à l’utilisation de **reactive** avec **reactiveValues**.

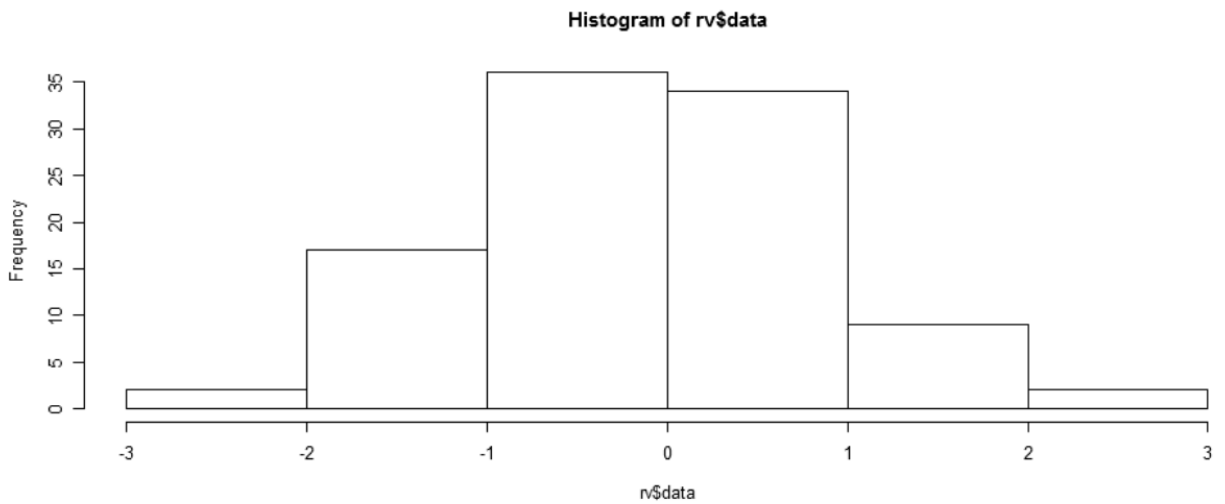
- **reactiveValues** permet d’initialiser une liste d’objets réactifs, un peu comme la liste des inputs
- On va pouvoir par la suite modifier la valeur des objets avec des **observe** ou des **observeEvent**

```
# server.R
rv <- reactiveValues(data = rnorm(100)) # init
# update
observeEvent(input$norm, { rv$data <- rnorm(100) })
observeEvent(input$unif, { rv$data <- runif(100) })
# plot
output$hist <- renderPlot({hist(rv$data)})
```

```
shinyApp(ui = fluidPage(
  actionButton(inputId = "norm", label = "Normal"),
  actionButton(inputId = "unif", label = "Uniform"),
  plotOutput("hist")
),
server = function(input, output) {
```

```
rv <- reactiveValues(data = rnorm(100))
observeEvent(input$norm, { rv$data <- rnorm(100) })
observeEvent(input$unif, { rv$data <- runif(100) })
output$hist <- renderPlot({ hist(rv$data) })
})
```

Normal Uniform



Normal Uniform

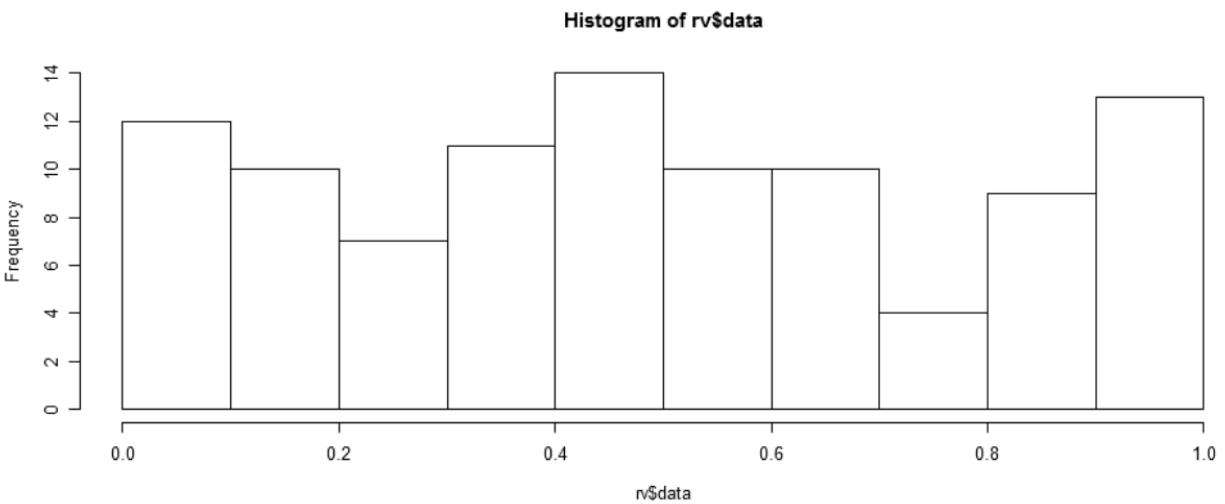


Figure 52:

12 Observe & fonctions d'update

12.1 Introduction

- Il existe une série de fonctions pour mettre à jour les inputs et certaines structures

- les fonctions commencent par `update...`
- On les utilise généralement à l'intérieur d'un `observe({expr})`
- La syntaxe est similaire à celle des fonctions de création
- **Attention** : il est nécessaire d'ajouter un argument `"session"` dans la définition du **server**

```
shinyServer(function(input, output, session) {...})
```

Sur des inputs :

- `updateCheckboxGroupInput`
- `updateCheckboxInput`
- `updateDateInput` `Change`
- `updateDateRangeInput`
- `updateNumericInput`
- `updateRadioButtons`
- `updateSelectInput`
- `updateSelectizeInput`
- `updateSliderInput`
- `updateTextInput`

Pour changer dynamiquement l'onglet sélectionné :

- `updateNavbarPage`, `updateNavlistPanel`, `updateTabsetPanel`

12.2 Exemple sur un input

```
shinyUI(fluidPage(
  titlePanel("Observe"),
  sidebarLayout(
    sidebarPanel(
      radioButtons(inputId = "id_dataset", label = "Choose a dataset", inline = TRUE,
        choices = c("cars", "iris", "quakes"), selected = "cars"),
      selectInput("id_col", "Choose a column", choices = colnames(cars)),
      textOutput(outputId = "txt_obs")
    ),
    mainPanel(fluidRow(
      dataTableOutput(outputId = "dataset_obs")
    ))
  )
))
```

```
shinyServer(function(input, output, session) {
  dataset <- reactive(get(input$id_dataset, "package:datasets"))

  observe({
    updateSelectInput(session, inputId = "id_col", label = "Choose a column",
      choices = colnames(dataset()))
  })
})
```

```

output$txt_obs <- renderText(paste0("Selected column : ", input$id_col))

output$dataset_obs <- renderDataTable(
  dataset(),
  options = list(pageLength = 5)
)
}
}

```

Observer

Choose a dataset

☒ cars ☐ iris ☐ quakes

Choose a column

speed

speed

dist

Show 10 entries

Search:

speed	dist
4	2
4	10
7	4
7	22
8	16

speed dist

Showing 1 to 5 of 50 entries

Previous 1 2 3 4 5 ...

10 Next

Figure 53: obs1

Observer

Choose a dataset

☐ cars ☒ iris ☐ quakes

Choose a column

Sepal.Length

Sepal.Length

Sepal.Width

Petal.Length

Petal.Width

Species

Show 10 entries

Search:

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa

Sepal.Length Sepal.Width Petal.Length Petal.Width Species

Showing 1 to 5 of 150 entries

Previous 1 2 3 4 5 ...

30 Next

Figure 54: obs2

12.3 Exemple sur des onglets

Il faut rajouter un id dans la structure

```
shinyUI(
  navbarPage(
    id = "idnavbar", # need an id for observe & update
    title = "A NavBar",
    tabPanel(title = "Summary",
      actionButton("goPlot", "Go to plot !")),
    tabPanel(title = "Plot",
      actionButton("goSummary", "Go to Summary !"))
  )
)
```

```
shinyServer(function(input, output, session) {
  observe({
    input$goPlot
    updateTabsetPanel(session, "idnavbar", selected = "Plot")
  })
  observe({
    input$goSummary
    updateTabsetPanel(session, "idnavbar", selected = "Summary")
  })
})
```

12.4 observeEvent

- Une variante de la fonction `observe` est disponible avec la fonction `observeEvent`
- On définit alors de façon explicite l'expression qui représente l'événement *et* l'expression qui sera exécutée quand l'événement se produit

```
# avec un observe
observe({
  input$goPlot
  updateTabsetPanel(session, "idnavbar", selected = "Plot")
})

# idem avec un observeEvent
observeEvent(input$goSummary, {
  updateTabsetPanel(session, "idnavbar", selected = "Summary")
})
```

13 Conditional panels

- Il est possible d'afficher conditionnellement ou non certains éléments :

```
conditionalPanel(condition = [...], )
```

- La condition peut se faire sur des inputs ou des outputs
- Elle doit être rédigée en **javascript**...

```
conditionalPanel(condition = "input.checkbox == true", [...])
```



```

library(shiny)
shinyApp(
  ui = fluidPage(
    fluidRow(
      column(
        width = 4,
        align = "center",
        checkboxInput("checkbox", "View other inputs", value = FALSE)
      ),
      column(
        width = 8,
        align = "center",
        conditionalPanel(
          condition = "input.checkbox == true",
          sliderInput("slider", "Select value", min = 1, max = 10, value = 5),
          textInput("txt", "Enter text", value = "")
        )
      )
    )
  ),
  server = function(input, output) {}
)

```

The screenshot displays the user interface of a Shiny application. It is divided into two horizontal sections. The top section, titled "Condition FALSE" in a red box, shows a checkbox labeled "View other inputs" which is unchecked. The bottom section, titled "Condition TRUE" in a green box, shows the same checkbox checked. To the right of the checkbox, a slider input labeled "Select value" is visible, with a range from 1 to 10 and a current value of 5. Below the slider is a text input field labeled "Enter text".

Figure 55: cond1

14 Débogage

14.1 Affichage console

- Un des premiers niveaux de débogage est l'utilisation de `print` console au-sein de l'application shiny.
- Cela permet d'afficher des informations lors du développement et/ou de l'exécution de l'application
- Dans **shiny**, on utilisera de préférence `cat(file=stderr(), ...)` pour être sûr que l'affichage marche dans tous les cas d'outputs, et également dans les logs avec **shiny-server**

```
output$distPlot <- renderPlot({
  x <- iris[, input$variable]
  cat(file=stderr(), class(x)) # affichage de la classe de x
  hist(x)
})
```

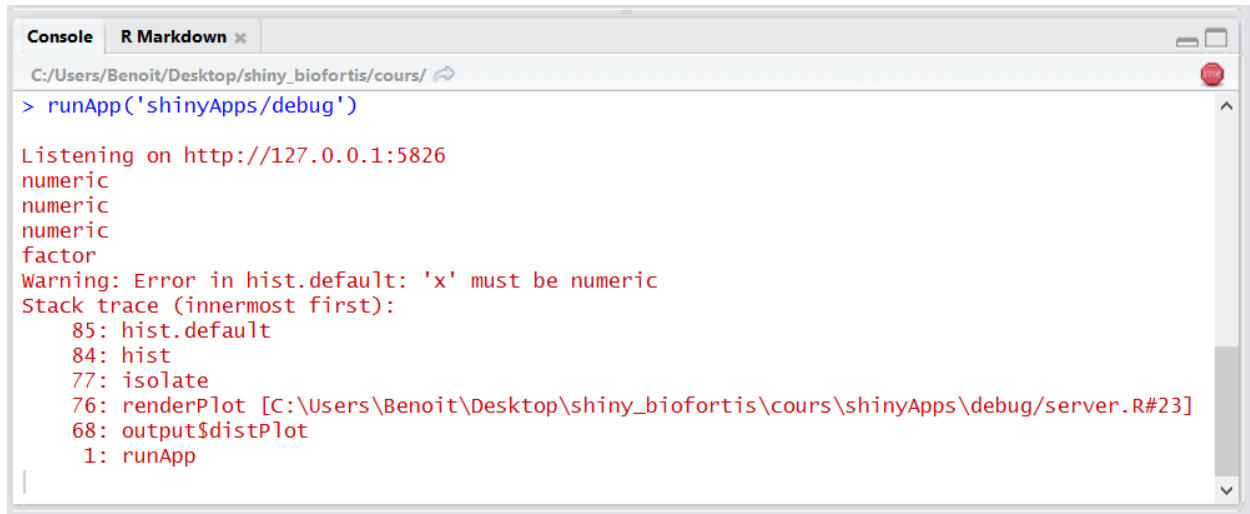


Figure 56: ggvis

14.2 Lancement manuel d'un browser

- On peut insérer le lancement d'un `browser()` à n'importe quel moment
- On pourra alors observer les différents objets et avancer pas-à-pas

```
output$distPlot <- renderPlot({
  x <- iris[, input$variable]
  browser() # lancement du browser
  hist(x)
})
```

- Ne pas oublier de l'enlever une fois le développement terminé...!

14.3 Lancement automatique d'un browser

- L'option `options(shiny.error = browser)` permet de lancer un `browser()` automatiquement lors de l'apparition d'une erreur

```
options(shiny.error = browser)
```

14.4 Mode "showcase"

- En lançant une application avec l'option `display.mode="showcase"` et l'utilisation de la fonction `runApp()`, on peut observer en direct l'exécution du code :

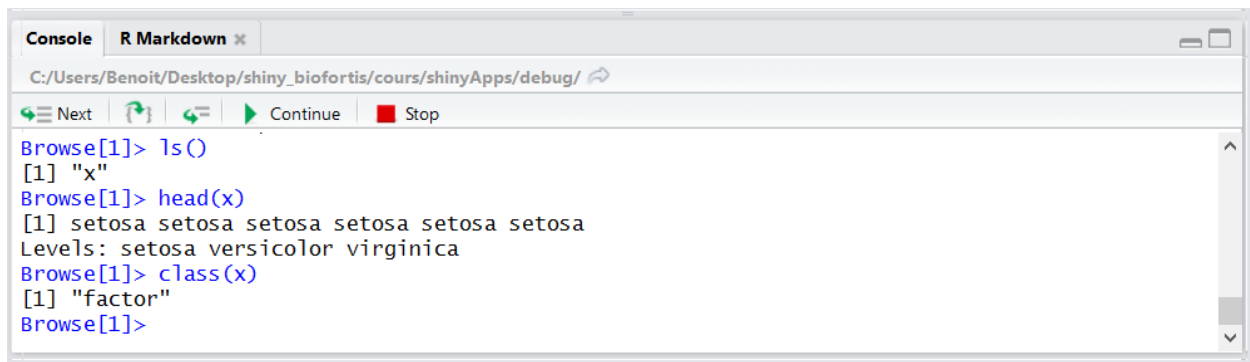


Figure 57: ggvis

```
runApp("path/to/myapp", display.mode="showcase")
```

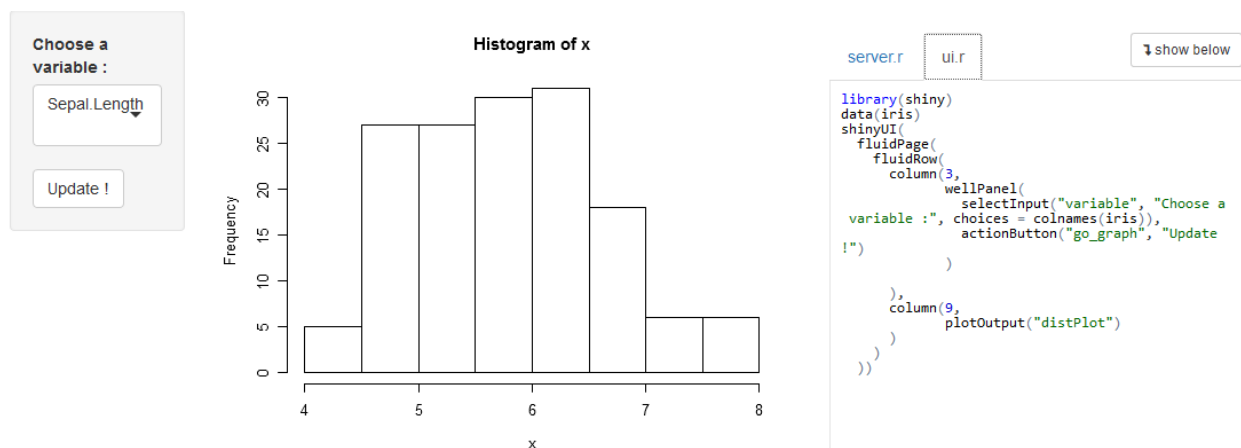


Figure 58: ggvis

14.5 Reactive log

- En activant l'option `shiny.reactlog`, on peut visualiser à tous instants les dépendances et les flux entre les objets réactifs de **shiny**
- soit en tapant `ctrl+F3` dans le navigateur web
- soit en insérant `showReactLog()` au-sein du code shiny

```

options(shiny.reactlog=TRUE)

output$distPlot <- renderPlot({
  x <- iris[, input$variable]
  showReactLog() # launch shiny.reactlog
  hist(x)
})

```

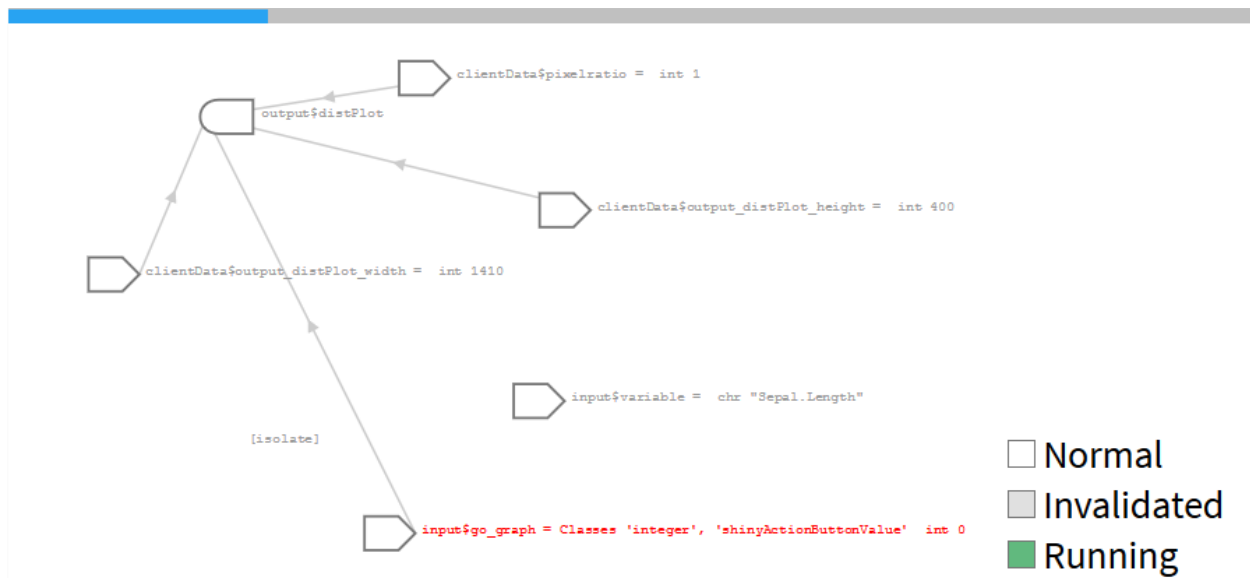


Figure 59: ggvis

14.6 Communication client/server

- Toutes les communications entre le client et le server sont visibles en utilisant l'option `shiny.trace`

```
options(shiny.trace = TRUE)
```

14.7 Traçage des erreurs

- Depuis `shiny_0.13.1`, on récupère la stack trace quand une erreur se produit
- Si besoin, on peut récupérer une stack trace encore plus complète, comprenant les différents fonctions internes, avec `options(shiny.fullstacktrace = TRUE)`

```
options(shiny.fullstacktrace = TRUE)
```

15 Quelques bonnes pratiques

- Préférer l'underscore (`_`) au point (`.`) comme séparateur dans le nom des variables. En effet, le `.` peut amener de mauvaises interactions avec d'autres langages, comme le **JavaScript**
- Faire bien attention à l'**unicité des différents identifiants** des inputs/outputs
- Pour éviter des problèmes éventuels avec **des versions différentes de packages**, et notamment dans le cas de **plusieurs applications shiny** et/ou différents environnements de travail, essayer d'utiliser `packrat`
- Mettre toute la **partie "calcul"** dans des **fonctions/un package** et effectuer des tests (`testthat`)
- Diviser la partie **ui.R** et **server.R** en plusieurs scripts, un par onglet par exemple :

```
Console R Markdown x
C:/Users/Benoit/Desktop/shiny_biofortis/cours/
> runApp('shinyApps/debug')

Listening on http://127.0.0.1:5826
SEND {"config":{"workerId":"","sessionId":"d881eec9a56887dd66d5d6bf2f8776ed"}}
RECV {"method":"init","data":{"go_graph:shiny.action":0,"variable":"Sepal.Length",".clientdata_output_distPlot_width":816,".clientdata_output_distPlot_height":400,".clientdata_output_distPlot_hidden":false,".clientdata_pixelratio":1,".clientdata_url_protocol":"http:",".clientdata_url_hostname":"127.0.0.1",".clientdata_url_port":"5826",".clientdata_url_pathname":"/",".clientdata_url_search":"",".clientdata_url_hash_initial":"",".clientdata_singletons":"",".clientdata_allowDataUriScheme":true}}
SEND {"custom":{"busy":"busy"}}
SEND {"custom":{"recalculating":{"name":"distPlot","status":"recalculating"}}}
SEND {"custom":{"recalculating":{"name":"distPlot","status":"recalculated"}}}
SEND {"custom":{"busy":"idle"}}
SEND {"errors":[],"values":{"distPlot":{"src":"data:image/png;base64 data","width":816,"height":400,"coordmap":[{"domain":{"left":3.84,"right":8.16,"bottom":-1.24,"top":32.24},"range":{"left":59.04,"right":785.76,"bottom":325.56,"top":58.04},"log":{"x":null,"y":null},"mapping":{}}]},"inputMessages":[]}}
RECV {"method":"update","data":{"variable":"Petal.Length"}}
```

Figure 60: ggvis

```
Console R Markdown x
C:/Users/Benoit/Desktop/shiny_biofortis/cours/
> runApp('shinyApps/debug')

Listening on http://127.0.0.1:5826
Warning: Error in hist.default: 'x' must be numeric
Stack trace (innermost first):
 88: h
 87: .handleSimpleError
 86: stop
 85: hist.default
 84: hist
 83: ..stacktraceon.. [C:\Users\Benoit\Desktop\shiny_biofortis\cours\shinyApps\debug\server.R#35]
 82: contextFunc
 81: env$runWith
 80: withReactiveDomain
 79: ctx$run
 78: ...
```

Figure 61: ggvis

```
# ui.R
shinyUI(
  navbarPage("Divide UI & SERVER",
    source("src/ui/01_ui_plot.R", local = TRUE)$value,
    source("src/ui/02_ui_data.R", local = TRUE)$value
  )
)

# server.R
shinyServer(function(input, output, session) {
  source("src/server/01_server_plot.R", local = TRUE)
  source("src/server/02_server_data.R", local = TRUE)
})
```

16 Quelques mots sur shiny-server

On peut déployer en interne nos applications shiny en installant un shiny-server.

- Uniquement sur linux : ubuntu 12.04+, RedHat/CentOS 5+, SUSE Enterprise Linux 11+
- Version gratuite : déployer plusieurs applications **shiny**
- Version payante :
 - authentification
 - ressources par applications (nombre de coeurs, mémoire, ...)
 - monitoring

Une fois le serveur installé, il suffit de déposer les applications dans le répertoire dédié, et elles deviennent directement accessibles via l'adresse *server:port_ou_redirection/nom_du_dossier*.



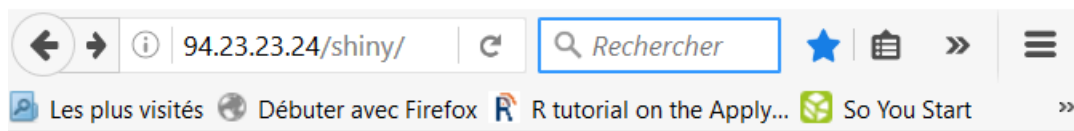
/srv/shiny-server/apps			
Nom	Ext	Taille	Date de modification
			08/10/2015 22:04:59
analysis			01/10/2015 16:14:47
demo_eric			08/10/2015 21:55:04
design			01/10/2015 16:58:07
Pelican			11/01/2016 16:48:49
Smart-Electric-Lyon			07/10/2015 15:15:23
Suivi-Commercial			07/01/2016 16:19:28
	.Rhistory	18 152 B	16/12/2015 18:12:25

Figure 62:

Des logs sont alors disponibles sous la forme de `print console` :

16.1 Références / Tutoriaux / Exemples

- <http://shiny.rstudio.com/>



Index of /apps/

- analysis/
- demo_eric/
- design/
- Pelican/
- Smart-Electric-Lyon/
- Suivi-Commercial/

Figure 63:

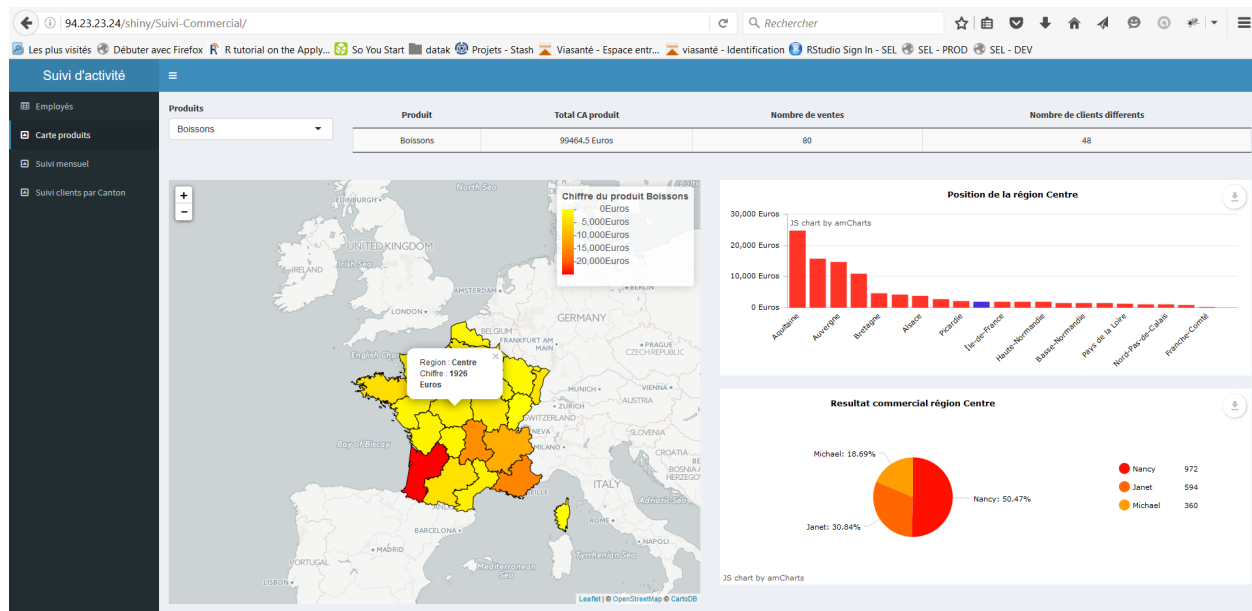


Figure 64:

/var/log/shiny-server			
Nom	Ext	Taille	Date de modification
..			29/03/2016 06:25:02
Suivi-Commercial-shiny-20160329-191039-52586	.log	1 590 B	29/03/2016 19:11:04
Suivi-Commercial-shiny-20151216-171623-50439	.log	1 290 B	16/12/2015 17:16:24
Suivi-Commercial-shiny-20151208-105155-32853	.log	1 290 B	08/12/2015 10:51:57
Suivi-Commercial-shiny-20151202-183808-54567	.log	1 290 B	02/12/2015 18:38:10
Suivi-Commercial-shiny-20151202-183751-39724	.log	1 290 B	02/12/2015 18:37:53
Suivi-Commercial-shiny-20151202-183719-59042	.log	1 290 B	02/12/2015 18:37:21
Suivi-Commercial-shiny-20151202-183523-41225	.log	1 290 B	02/12/2015 18:35:24
Suivi-Commercial-shiny-20151202-183307-50815	.log	782 B	02/12/2015 18:33:09
Suivi-Commercial-shiny-20151202-183258-50423	.log	782 B	02/12/2015 18:32:59
Suivi-Commercial1-shiny-20151208-104753-50451.l...		1 290 B	08/12/2015 10:47:55

Figure 65:

- <http://shiny.rstudio.com/articles/>
- <http://shiny.rstudio.com/tutorial/>
- <http://shiny.rstudio.com/gallery/>
- <https://www.rstudio.com/products/shiny/shiny-user-showcase/>
- <http://www.showmeshiny.com/>