

Présentation R - Shiny

Jeffery P., jeffery.petit@datastorm.fr
Benoit T., benoit.thieurmél@datastorm.fr

Table des matières

1	Shiny : créer des applications web avec le logiciel R	1
2	Créer une première application avec shiny	2
3	Interactivité et communication	3
4	Les inputs	10
5	Outputs	15
6	Structurer sa page	19
7	Customisation avec des propriétés CSS	25
8	Les atouts d'une application moderne : graphiques interactifs	27
9	Isolation	30
10	Principe des expressions réactives	31
11	Observe & fonctions d'update	34
12	Affichage conditionnel : « conditionalPanel »	37
13	Débogage	38
14	Quelques bonnes pratiques	42
15	Quelques mots sur shiny-server	42

1 Shiny : créer des applications web avec le logiciel R

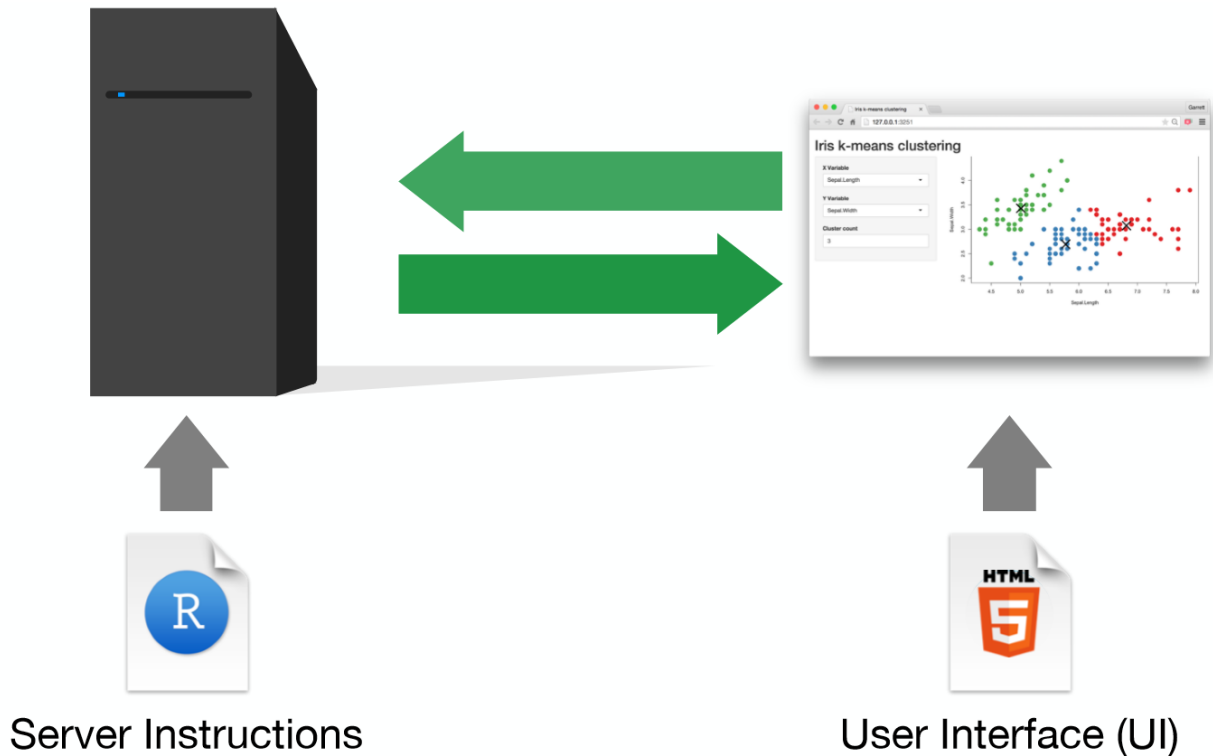
Shiny est un package **R** qui permet la création simple d'applications web interactives depuis le logiciel open-source **R**.

Il combine les avantages suivants :

- Pas de connaissances *web* nécessaires.
- Pouvoir de calcul de R et l'interactivité du web actuel.
- Applications locales ou partagées avec l'utilisation d'un **shiny-server**.

Plus de détails sur :

- **Shiny** : <http://shiny.rstudio.com>.
- L'utilisation de **shiny-server** : <https://www.rstudio.com/products/shiny/shiny-server/>.



© CC 2015 RStudio, Inc.

FIGURE 1 – Serveur et communication

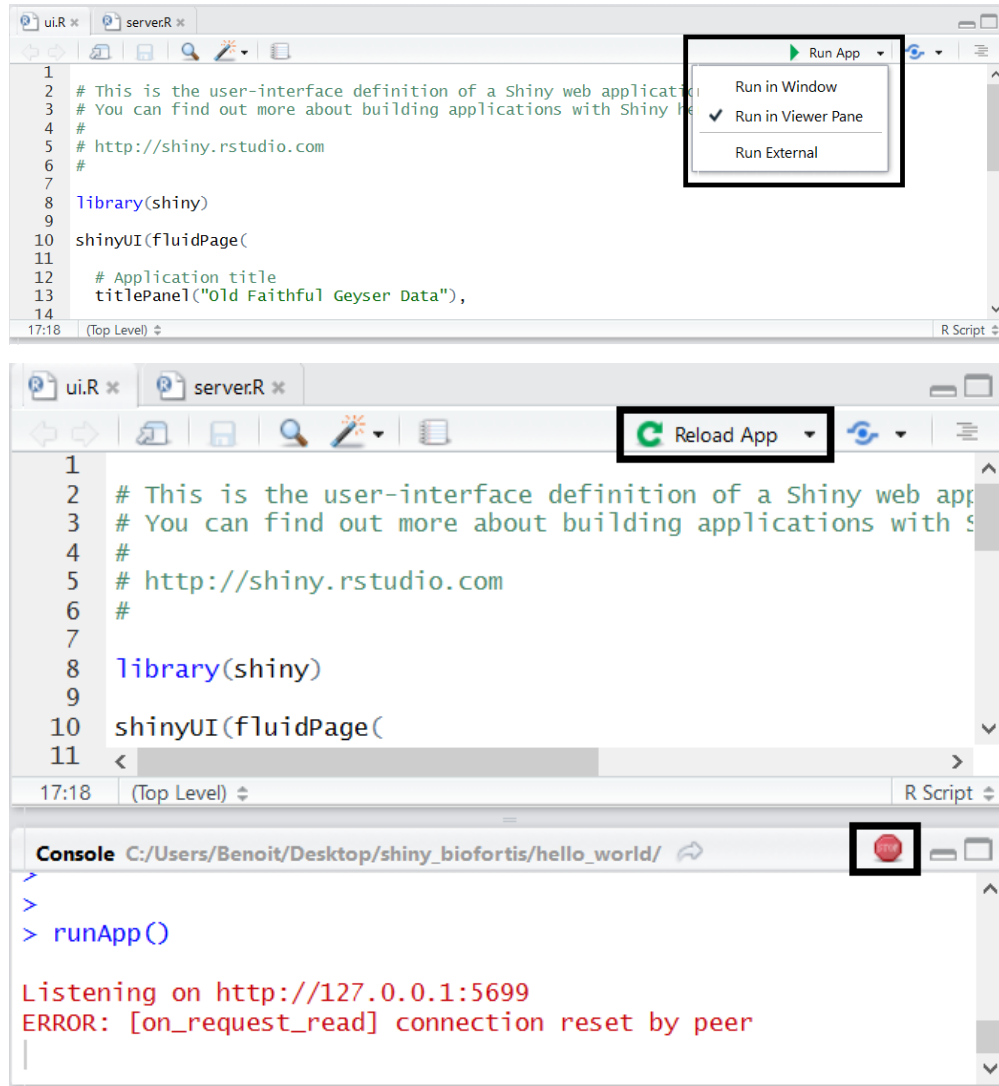
Comme illustré sur la figure 1, une application **shiny** nécessite un ordinateur/un serveur exécutant des instructions **R**. La communication avec l'interface utilisateur se fait par l'intermédiaire d'identifiants.

2 Créer une première application avec shiny

- Initialiser une application est simple avec **RStudio**, en créant un nouveau projet
 - File > New Project > New Directory > Shiny Web Application
- L'application créée :
 - Repose sur deux scripts : **ui.R** et **server.R**, ou un seul : **app.R**
 - Utilise par défaut le *sidebar layout* (détaillé plus tard)

Lors du développement, R Studio fournit des boutons utiles :

- Lancement de l'application : bouton **Run app**
 - « 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
- Actualisation : bouton **Reload app**
- Arrêt : bouton **Stop**



3 Interactivité et communication

3.1 Structure d'une application

Lorsque vous créez un projet Shiny depuis R Studio, vous obtenez soit un script nommé *app.R* soit deux scripts nommés *ui.R* et *server.R*. Dans les deux cas, l'application résultante est la même (cf. figure 2, page 4), il s'agit de deux manières différentes de développer :

- Mono-script *app.R* : on définit deux objets, le script se termine par l'instruction `shiny::shinyApp(ui, server)`
- Bi-script *ui.R* et *server.R* (conseillé) : on définit les objets sus-mentionnés dans deux scripts distincts, on lance l'application avec l'instruction `shiny::runApp(appDir)`

Hello Shiny!

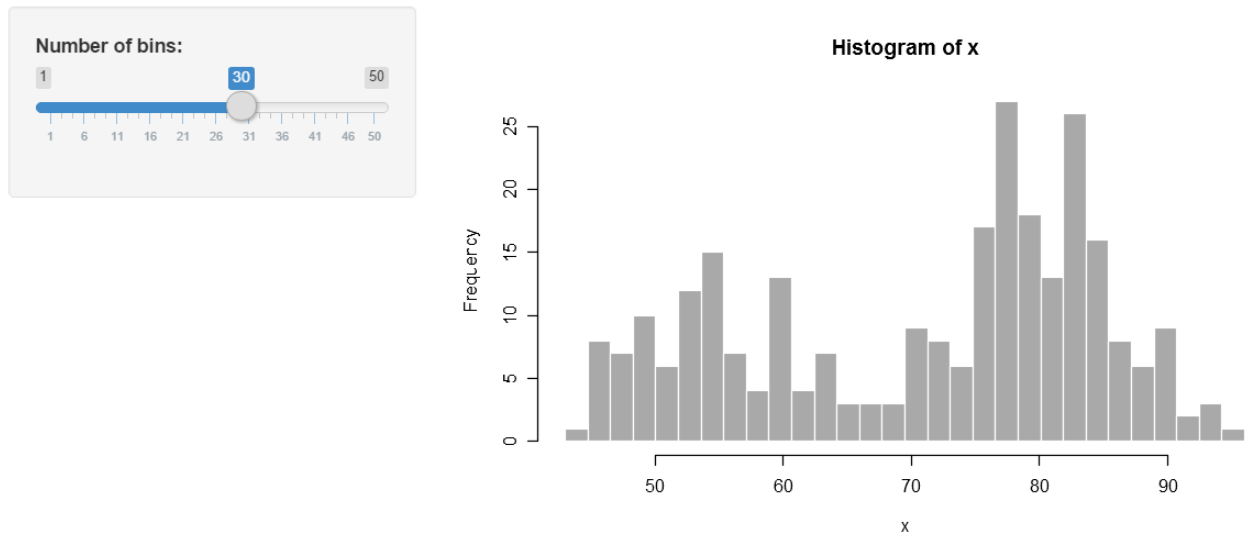
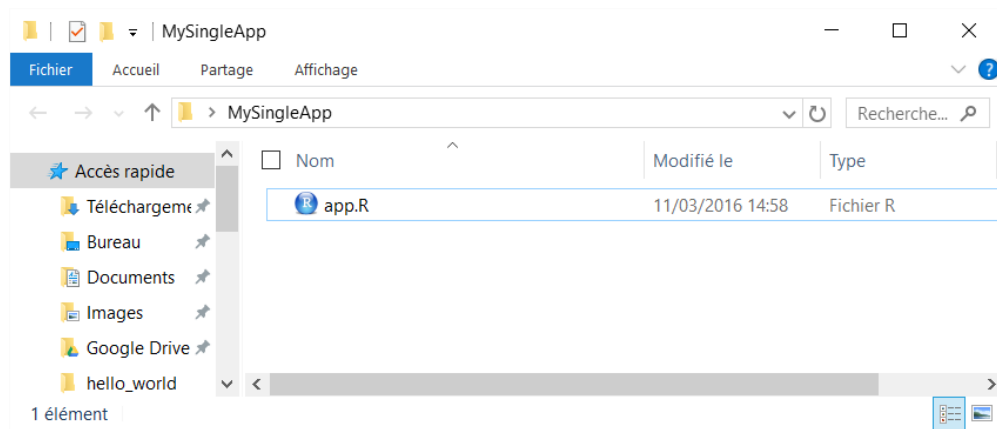


FIGURE 2 – Application de base : « Hello shiny! »

3.1.1 Un seul fichier

- Enregistré sous le nom **app.R**
- Se terminant par la commande `shinyApp()`
- 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({
```

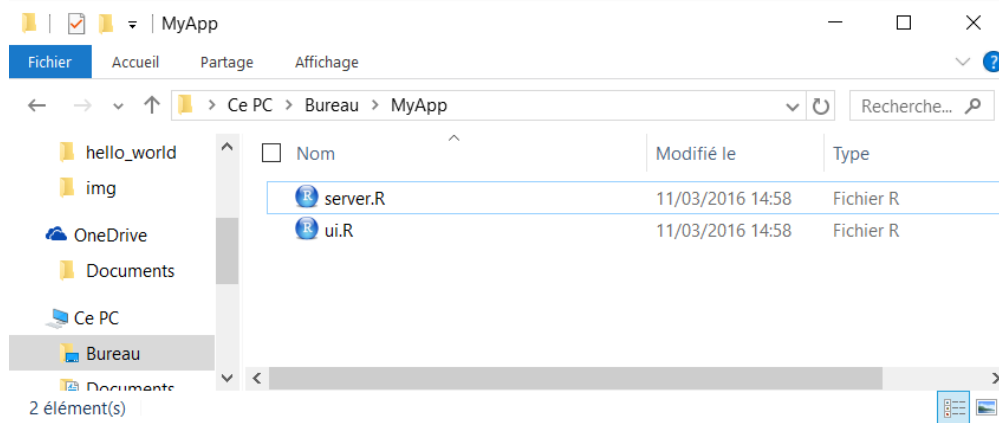
```

    hist(rnorm(input$num))
  })
}
shinyApp(ui = ui, server = server)

```

3.1.2 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

```

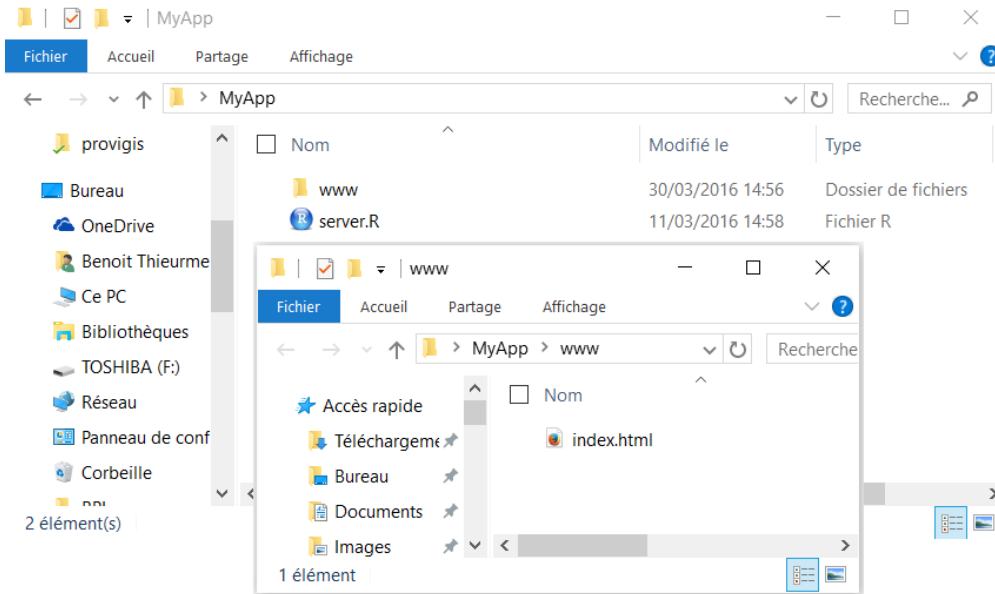
library(shiny)
function(input, output) {
  output$hist <- renderPlot({hist(rnorm(input$num))})
}

```

3.1.3 UI en HTML

Même si en général on code l'interface dans un script **R**, il est également possible de le coder entièrement dans un fichier **.html**. Plus d'informations à cette adresse <http://shiny.rstudio.com/articles/html-ui.html>

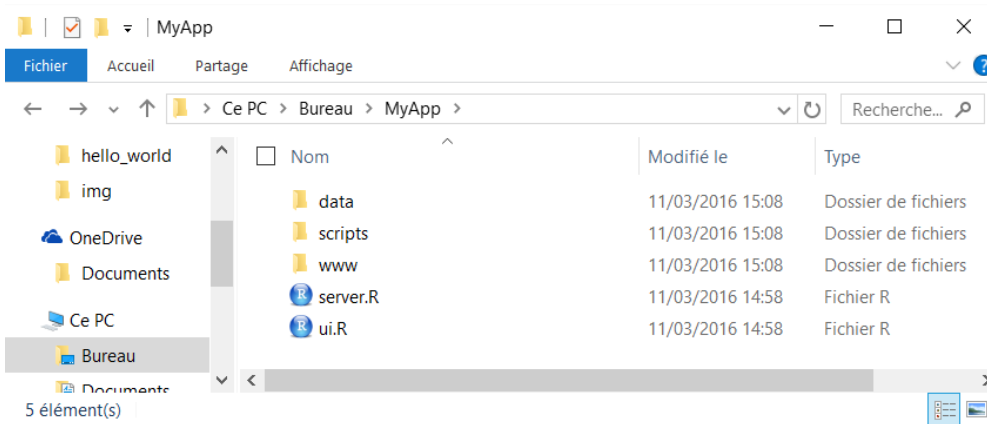
- Côté interface utilisateur, un fichier **index.html** dans le répertoire **www**
- Côté serveur dans le script **server.R**



3.1.4 Données/fichiers complémentaires

Côté serveur, la session s'exécute dans le répertoire où se trouvent les scripts **R** de l'application (en général `ui.R` et `server.R`). On peut donc accéder de façon relative à tous les objets présents dans le dossier de l'application.

Côté interface, l'application web, comme de convention, accède à tous les éléments présents dans le dossier `www`



3.1.5 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

3.1.6 Exemple de présentation

Dans la suite, on se pose dans le cas d'une application *duale*

ui.R :

```
library(shiny)
shiny::run
# 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
  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')
```

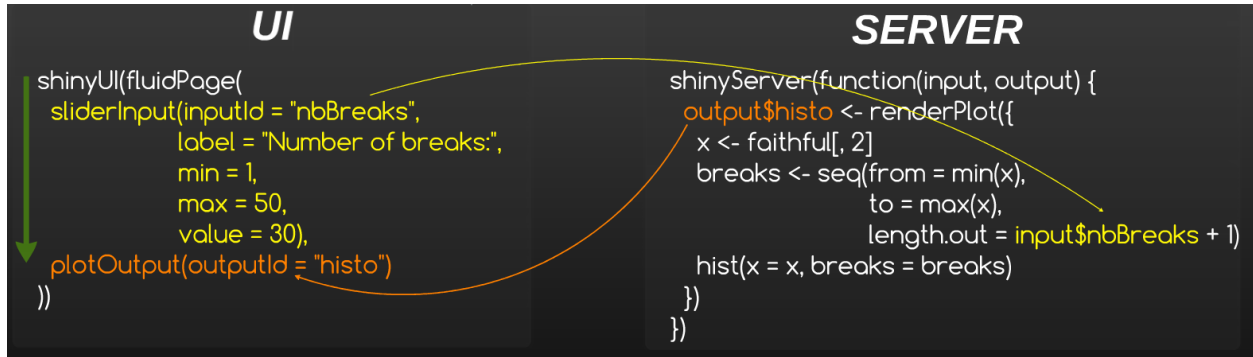


FIGURE 3 – Dialogue « Interface <> Serveur »

```

})
})

```

Cet exemple simple nous permet de déduire le fonctionnement suivant :

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

Nous illustrons sur la figure 3, page 8, la communication entre l'interface et le server grâce aux identifiants.

Quelques remarques importantes :

- « Interface - **ui** » et « Serveur - **server** » 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

3.2 Principes fondamentaux

La définition de l'interface utilisateur : **ui.R**

- Déclaration des inputs
- Placement des outputs
- Ajout de textes, descriptions, images, etc.

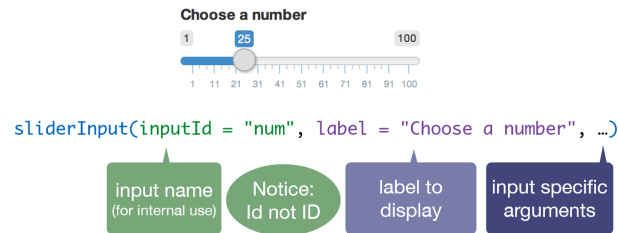
La partie serveur/calculs : **server.R**

- Déclaration et calcul des outputs.
- Manipulation des données sous conditionnement d'input(s).
- Codage **R** standard, etc.

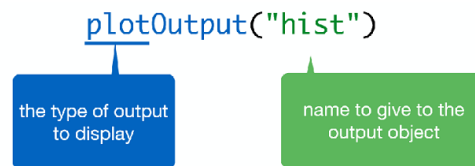
3.3 Interface : ui.R

Deux types d'éléments

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



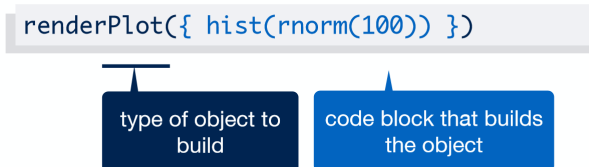
- `xxOutput(ouputId = ...)` :
 - fait référence à un output créé et défini côté serveur ;
 - en général : graphiques et tableaux.



3.4 Serveur : server.R

Définition des outputs dans le serveur

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



4 Les inputs

Buttons



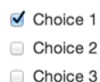
`actionButton()`
`submitButton()`

Single checkbox



`checkboxInput()`

Checkbox group



`checkboxGroupInput()`

Date input



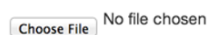
`dateInput()`

Date range



`dateRangeInput()`

File input



`fileInput()`

Password Input

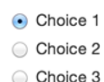


`passwordInput()`

accompany other widgets.

`numericInput()`

Radio buttons



`radioButtons()`

Select box



`selectInput()`

Sliders



`sliderInput()`

Text input



`textInput()`

Dans la suite, nous donnons une liste non exhaustive de widgets disponibles. Il s'agit d'éléments de base très largement utilisés par les développeurs dans le cadre d'applications simples ou complexes.

4.1 Valeur numérique

— La fonction :

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

— Exemple :

```
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)
```



4.2 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"
```

<div style="border: 1px solid #ccc; padding: 5px; width: 180px;"> <p>Enter a text</p> <input style="width: 150px;" type="text" value="test"/> </div>	<p>Value: [1] "test"</p> <p>Class: character</p>
--	--

4.3 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)
```

<div style="border: 1px solid #ccc; padding: 5px; width: 180px;"> <p>Select among the list:</p> <div style="border: 1px solid #ccc; padding: 2px;">3</div> </div>	<p>Value: [1] "3"</p> <p>Class: character</p>
<div style="border: 1px solid #ccc; padding: 5px; width: 180px;"> <p>Select among the list:</p> <div style="border: 1px solid #ccc; padding: 2px;"> Third Second </div> </div>	<p>Value: [1] "3" "2"</p> <p>Class: character</p>

4.4 Checkbox

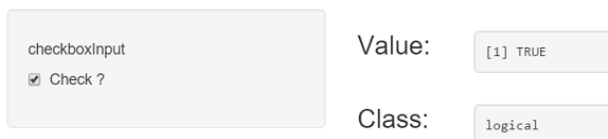
— La fonction :

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

— Exemple :

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

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



The image shows a Shiny checkbox input. On the left, a light gray box contains the text "checkboxInput" and a checked checkbox followed by "Check ?". To the right, the "Value:" field displays "[1] TRUE" and the "Class:" field displays "logical".

4.5 Choix multiples


— 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
```



The image shows a Shiny checkbox group input. On the left, a light gray box contains the text "Please select" and three checked checkboxes labeled "First", "Second", and "Third". To the right, the "Value:" field displays "[1] \"2\" \"3\"" and the "Class:" field displays "character".

4.6 Radio boutons

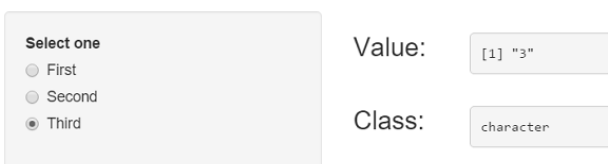
— La fonction :

```
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"
```



The image shows a Shiny radio button input. On the left, a light gray box contains the text "Select one" and three radio buttons labeled "First", "Second", and "Third", with the "Third" button selected. To the right, the "Value:" field displays "[1] \"3\"" and the "Class:" field displays "character".

4.7 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"

The screenshot shows a user interface element titled "Please enter a date" with a text input field containing "07/12/2015". To the right, there are two labels: "Value:" and "Class:". The "Value:" label points to a box containing "[1] \"2015-12-07\"", and the "Class:" label points to a box containing "Date".

4.8 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

The screenshot shows a user interface element titled "Please Select a date range" with a date range picker showing "2015-01-01" to "2015-08-12". To the right, there are two labels: "Value:" and "Class:". The "Value:" label points to a box containing "[1] \"2015-01-01\" \"2015-08-12\"", and the "Class:" label points to a box containing "Date".

4.9 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)
```



4.10 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)
```



4.11 Importer un fichier

— La fonction :

```
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
```

Select a file

Choisissez un fichier tab2.csv

Upload complete

Value:

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

4.12 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"
```

Action button

Click !

Value:

Class:

4.13 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. Un tutoriel est disponible à l'adresse suivante <http://shiny.rstudio.com/articles/building-inputs.html>

De même, deux applications sont données à titre d'exemples :

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

5 Outputs

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

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})
```

5.1 Print

- ui.r :

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

- server.r :

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



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

5.2 Text

- ui.r :

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

- server.r :

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

Hello shiny !

5.2.1 Plot

- ui.r :

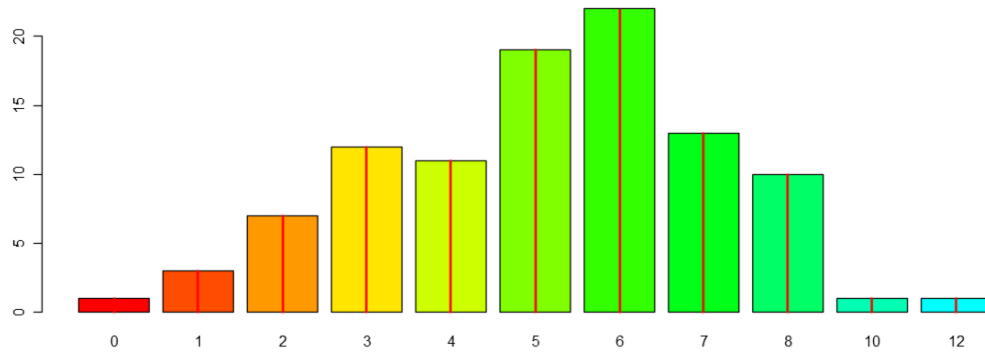
```
plotOutput("myplot")
```



```

— server.r :
output$myplot <- renderPlot({
  hist(iris$Sepal.Length)
})

```



5.3 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

5.4 DataTable

```

— ui.r :
dataTableOutput(outputId = "dataTable")

— server.r :

```

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

Show 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

Showing 1 to 5 of 5 entries

5.5 Définir des éléments de l'UI côté SERVER

5.5.1 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 dans un tableau chargé dans le server. Cela est possible avec les fonctions `uiOutput` et `renderUI`.

5.5.2 Exemple simple

— `ui.r` :

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

— `server.r` :

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

dataset :

faithful ▼

Column

eruptions ▲

eruptions

waiting

dataset :

iris ▼

Column

Sepal.Length ▲

Sepal.Length

Sepal.Width

Petal.Length

5.5.3 Exemple plus complexe

On peut également renvoyer un élément plus complexe de l'UI, par exemple tout un `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")))
  )
)
```

5.6 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 (<https://d3js.org/>). Un exemple est disponible dans le dossier `shinyApps/build_output`.

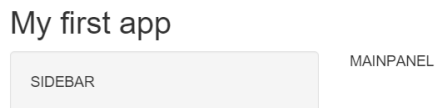
6 Structurer sa page

6.1 Organisation 1/3 - 2/3 : « 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")
  )
))
```

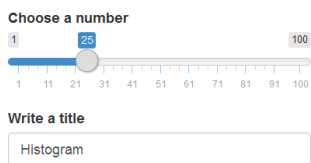


6.2 Contenu grisé : « wellPanel »

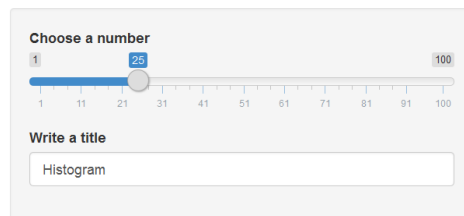
Comme avec le `sidebarPanel` précédent, on peut griser un ensemble d'éléments en utilisant un `wellPanel` :

```
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")
))
```

Without wellPanel



With wellPanel



6.3 Page à onglets : « navbarPage »

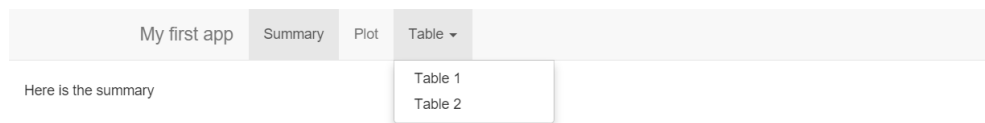
Utiliser une barre de navigation et des onglets avec `navbarPage` et `tabPanel` :

```
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"))
  )
)
```

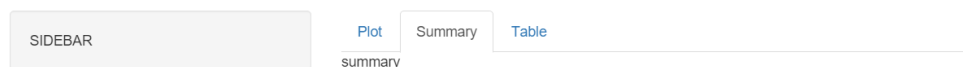


6.4 Organisation du contenu en onglets : « `tabsetPanel` »

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

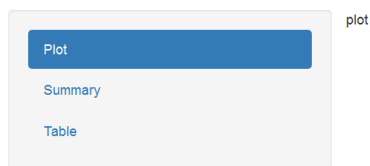
My first app



6.5 Onglets verticaux : “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"))
  )
))
```



6.6 Diviser pour mieux organiser : “Grid Layout”

Créer sa propre organisation avec `fluidRow()` et `column()`. Une seule règle : **chaque ligne peut être divisée en 12 colonnes**.

Le dimensionnement final de la page est automatique en fonction des éléments dans les lignes / colonnes

Ce principe, très simple, est à respecter avec rigueur (et indentation de préférence) :

```
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"),
  )
)
```

My first app	Summary	Plot	Table
column 1	column 2	column 3	

6.7 Inclure des balises HTML

De nombreuses 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"       "eventsources" "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"
```

```
tags$a(href = "www.rstudio.com", "RStudio")
```

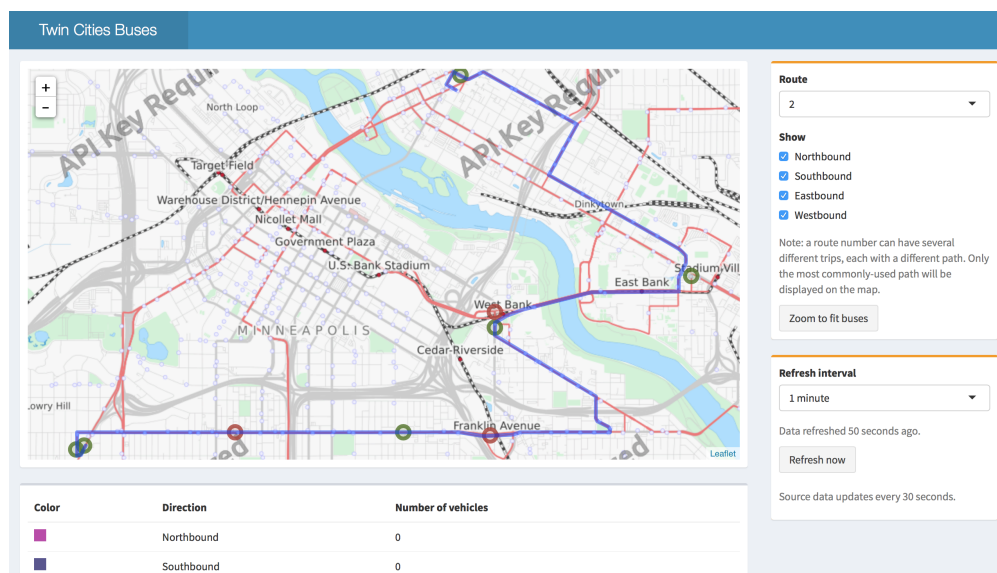


C'est également possible de passer du code **HTML** directement en utilisant la fonction du même nom :

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

6.8 Créer facilement des tableaux de bord : “shinydashboard”

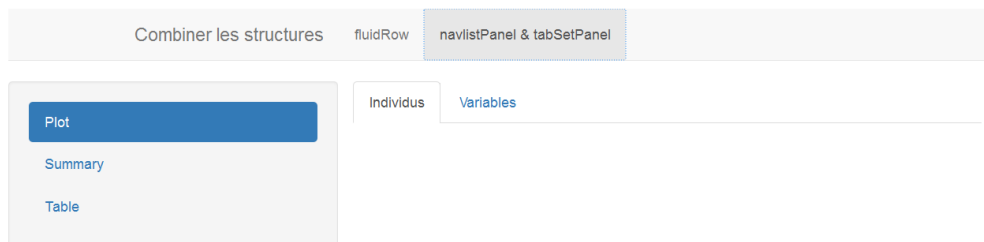
Le package “shinydashboard” (<https://rstudio.github.io/shinydashboard/>) propose d'autres fonctions pour créer des tableaux de bord :



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

6.9 Combiner les structures

Toutes les structures peuvent s'utiliser en même temps !



7 Customisation avec des propriétés CSS

Shiny utilise Bootstrap pour la partie CSS (Custom StyleSheet). À l'image du développement web « classique », nous pouvons modifier la feuille de style 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 l'en tête HTML ;
- en écrivant individuellement du CSS aux éléments.

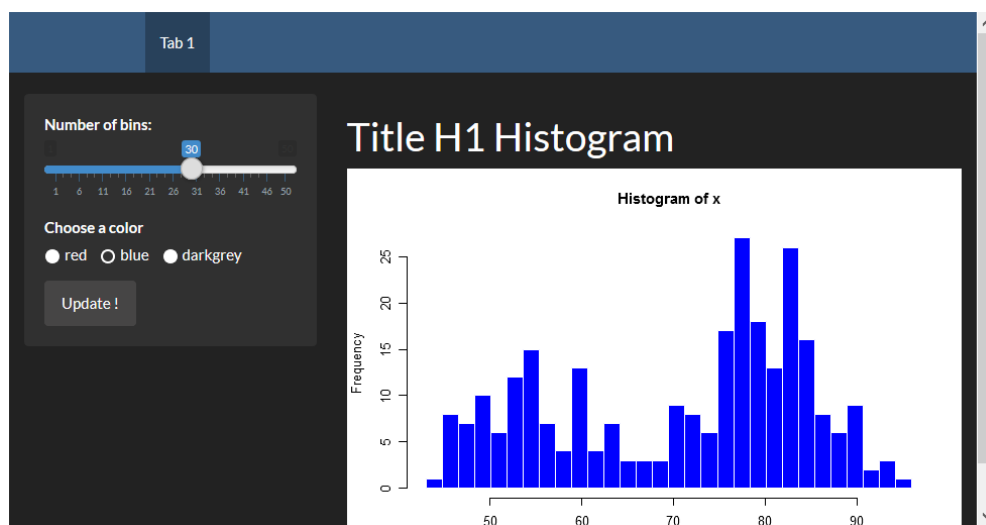
Il y a une notion d'ordre et de priorité sur ces trois informations : le CSS « individuel » l'emporte sur le CSS du header, qui l'emporte sur le CSS externe.

NB : On peut aussi utiliser le package *shinythemes* (<http://rstudio.github.io/shinythemes>).

7.1 Avec un fichier `.css` externe

On peut par exemple télécharger un thème sur bootswatch. Il y a deux façons pour le renseigner dans l'application : soit avec argument `theme` dans `fluidPage`, soit avec une balise HTML (`tags$head` et/ou `tags$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
)
```

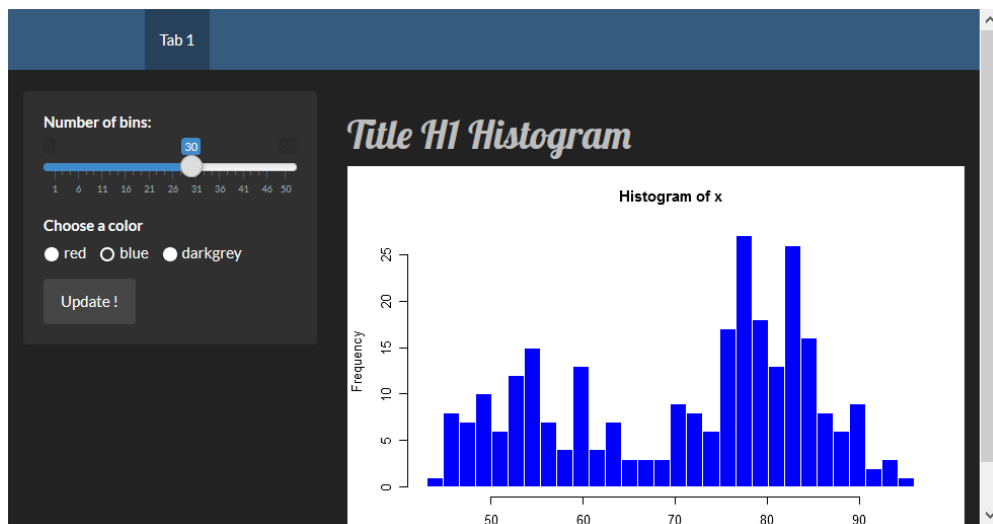


7.2 Ajout de propriétés dans l'en-tête

Les propriétés CSS incluses dans l'en-tête seront prioritaires sur celles présentes dans le fichier CSS externe. On utilise pour cela les tags HTML : `tags$head` et `tags$style`.

Il est conseillé de ne pas surcharger l'en-tête pour préserver la lisibilité et le débogage (et la factorisation). Ce genre d'ajout est recommandé uniquement en cas de modification ponctuelle et non pour définir une charte graphique complète.

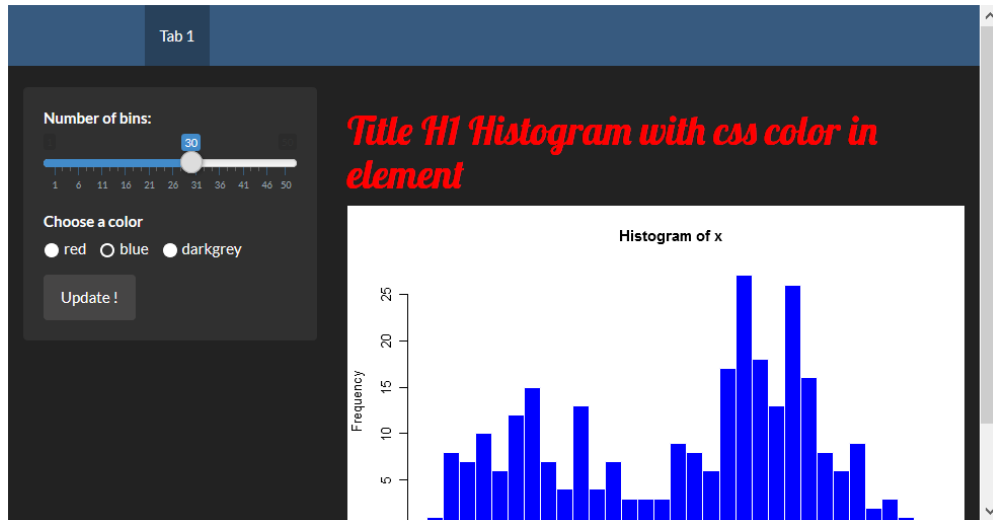
```
library(shiny)
tags$head(
  tags$style(HTML("h1 { color: #48ca3b;}"))
)
```



7.3 Modifier les propriétés d'un élément unique

Pour finir, on peut également passer directement des propriétés CSS aux éléments HTML :

```
library(shiny)
h1("Mon titre", style = "color: #48ca3b;")
```



8 Les atouts d'une application moderne : graphiques interactifs

Avec notamment l'arrivée du package *htmlwidgets* (<http://www.htmlwidgets.org/>), de plus en plus de fonctionnalités tirées des bibliothèques Javascript sont accessibles sous **R** :

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

De nombreux autres exemples sont disponibles dans la galerie suivante <http://gallery.htmlwidgets.org/>

8.1 Utilisation dans shiny

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

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

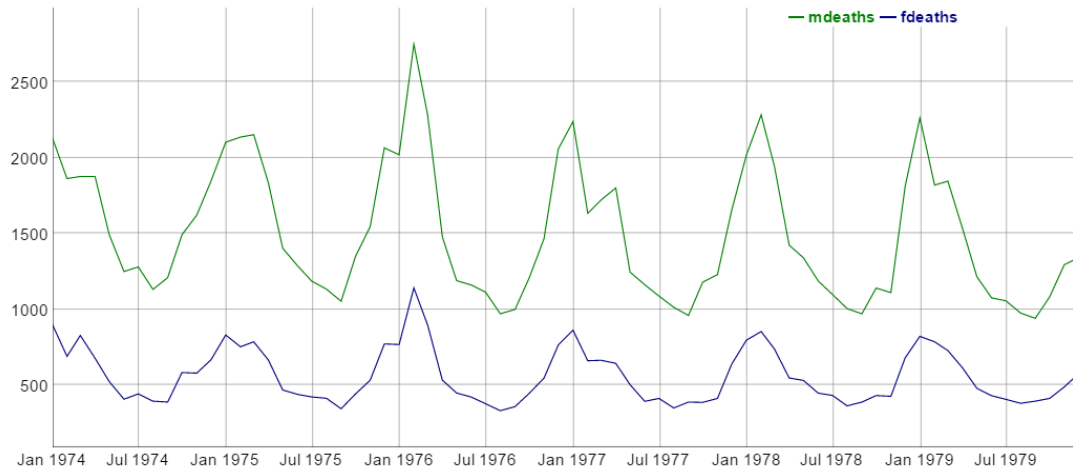
Par exemple avec le package *dygraphs* :

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

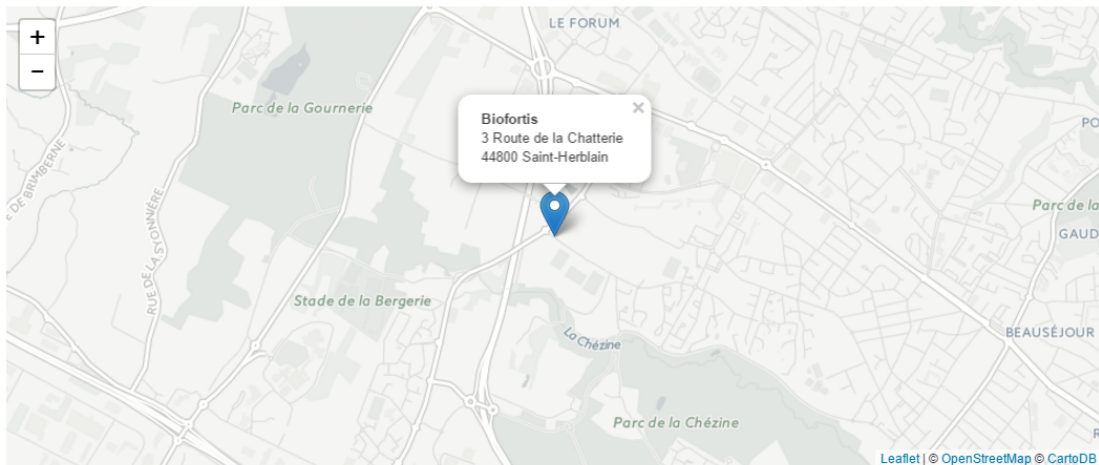
```
# Ui
```

```
dygraphOutput("dygraph")
```

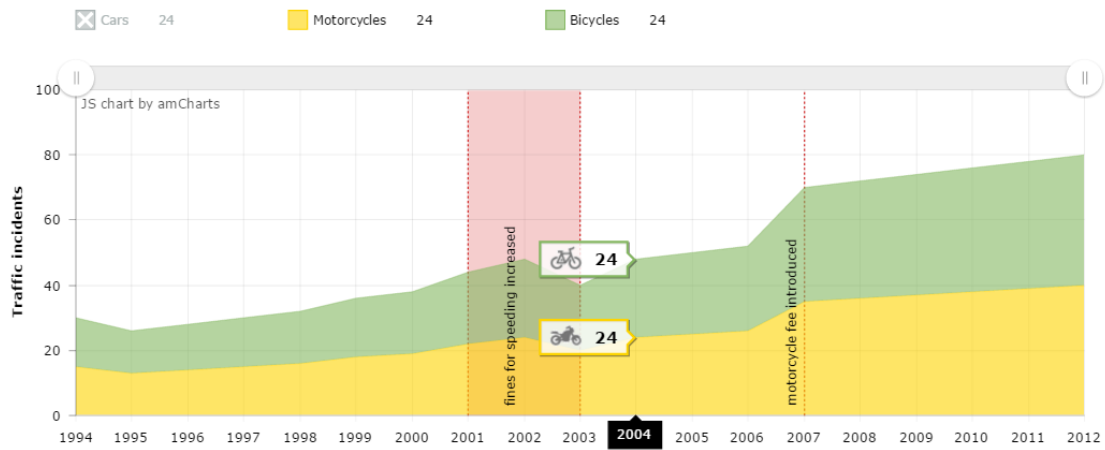
dygraphs



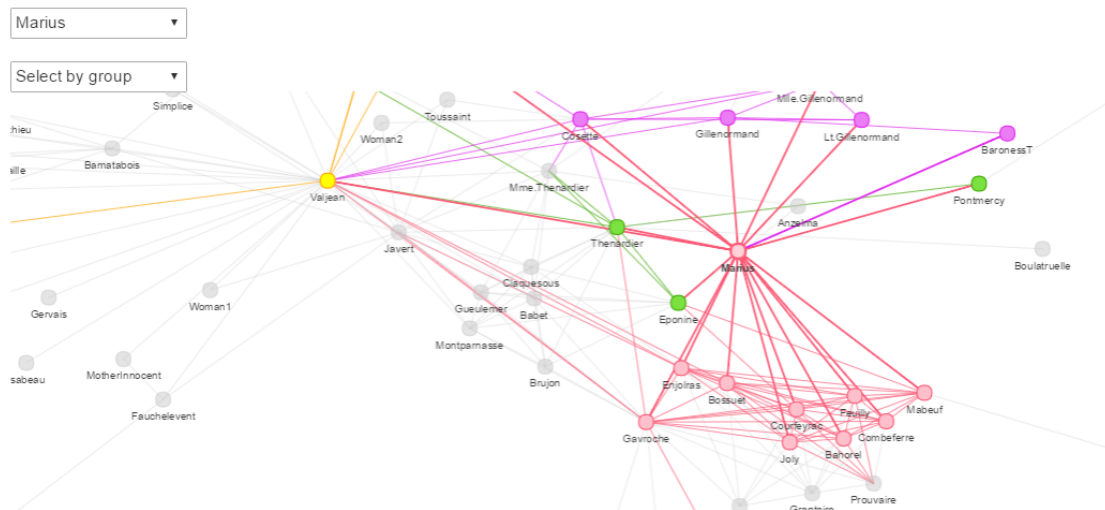
leaflet



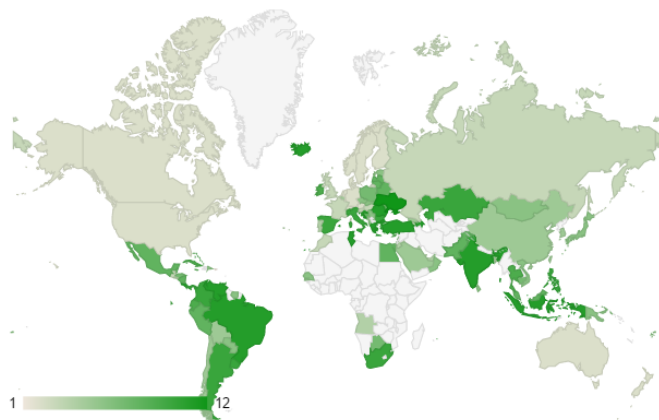
rAmCharts



visNetwork



googleVis Example



Data from:

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

9 Isolation

9.1 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 plus finement ce rafraichissement. Par exemple, pour utiliser un bouton de validation (**actionButton**) afin de déclencher la mise à jour de l'interface.

- Un **input** peut être isolé comme cela `isolate(input$id)`
- Un **bloc d'instructions** peut être isolé avec la notation suivante `isolate({expr})` et l'utilisation de `{}`

9.2 Exemple 1

- **ui.r** :

Trois inputs : **color** et **bins** pour l'histogramme, et un **actionButton** :

```
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"))
  )
))
```

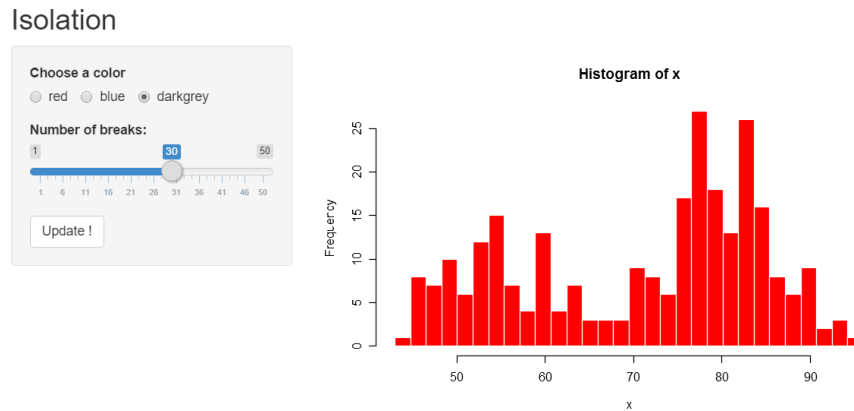
- **server.r** :

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.



9.3 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 mis à jour quand l'utilisateur cliquera sur le bouton **ou** quand la couleur changera.

10 Principe des 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.

10.1 Exemple sans 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))
})
```

10.2 Exemple avec expression réactive

- **server.R** : le calcul n'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))
})
```

10.3 Quelques remarques

- 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>).

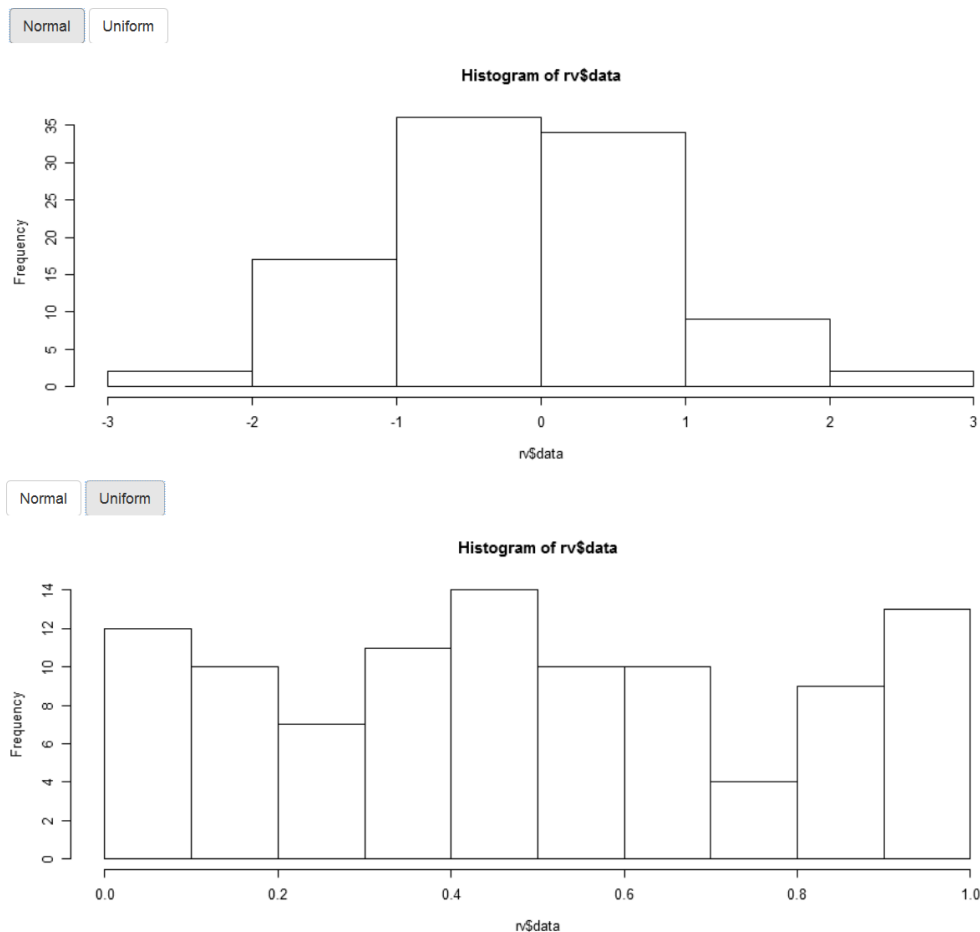
- Elle se comporte comme un output standard : mis à jour chaque fois qu'un input présent dans le code change.
- Elle se comporte comme un input dans un *renderXX* : l'output est mis à jour quand l'expression réactive change.
- On récupère sa valeur comme un appel à une fonction, avec des parenthèses d'appel « `mon_objet_reactif()` ».

10.4 Objets réactifs : « `reactiveValues` »

Il existe une alternative à l'utilisation de `reactive()` avec `reactiveValues()`. Cette fonction permet d'initialiser une liste d'objets réactifs, un peu comme la liste des inputs. On va ensuite pouvoir 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) })
})
```



11 Observe & fonctions d'update

11.1 Introduction

Il existe une série de fonctions pour mettre à jour les inputs et certaines structures, les fonctions commencent par `updateXX`. On les utilise généralement à l'intérieur d'un `observe({expr})` dont la syntaxe est similaire à celle des fonctions de création.

Attention : il est nécessaire d'ajouter un argument **session** dans la définition de la fonction du serveur.

```
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`

11.2 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
 dist

Show entries Search:

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

Showing 1 to 5 of 50 entries

Previous **1** 2 3 4 5 ...

10 Next

Observer

Choose a dataset
☐ cars ☒ iris ☐ quakes

Choose a column

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

Show 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

Showing 1 to 5 of 150 entries

Previous **1** 2 3 4 5 ...

30 Next

11.3 Exemple sur des onglets

Il faut rajouter un identifiant dans la structure (attribut id).

```
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")
  })
})
```

11.4 Observateur : « *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")
})
```

12 Affichage conditionnel : « *conditionalPanel* »

— 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) {}
)

```

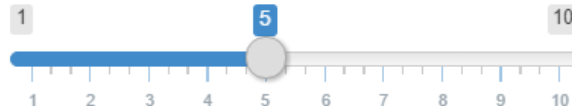
Condition FALSE

☐ View other inputs

Condition TRUE

☒ View other inputs

Select value



Enter text

13 Débogage

13.1 Affichage console

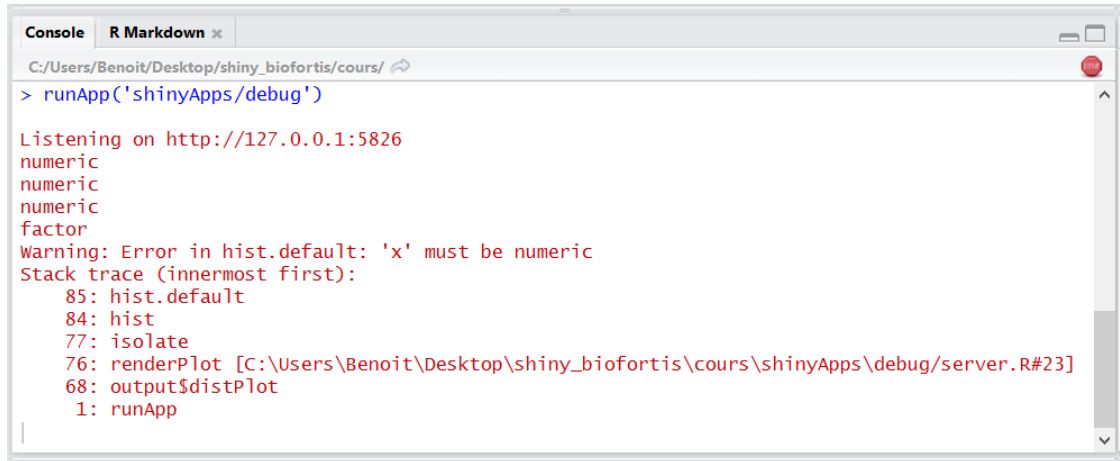
Un des premiers niveaux de débogage est l'utilisation de `print` console au sein de l'application shiny. Bien que déconseillé, cela permet d'afficher des informations lors du développement et/ou de l'exécution de l'application. Dans **shiny**, on utilisera de préférence `cat(file=stderr(), ...)` pour 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)
})
```



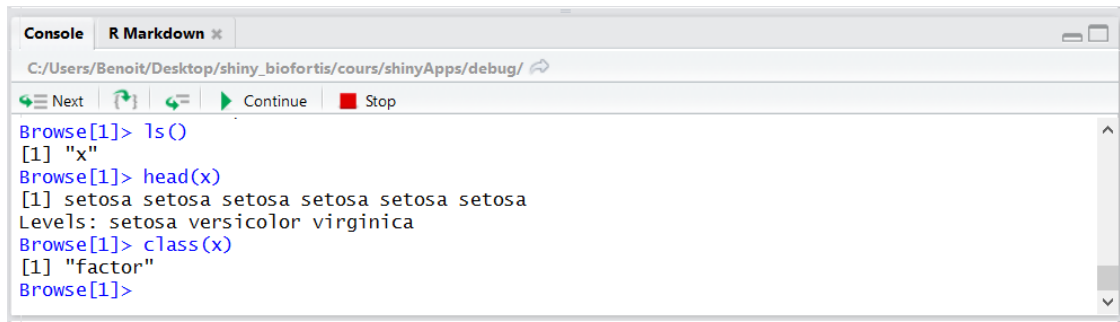
NB : bien que largement répandue, nous recommandons de substituer cette pratique par l'utilisation de point(s) d'arrêt(s) lorsqu'il s'agit de rechercher des anomalies lors du développement. Lorsque l'instruction sera exécutée, l'application s'arrêtera et vous serez renvoyé à la console **R** où il sera possible d'inspecter tous les objets disponibles...

13.2 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... !



13.3 Lancement automatique d'un browser

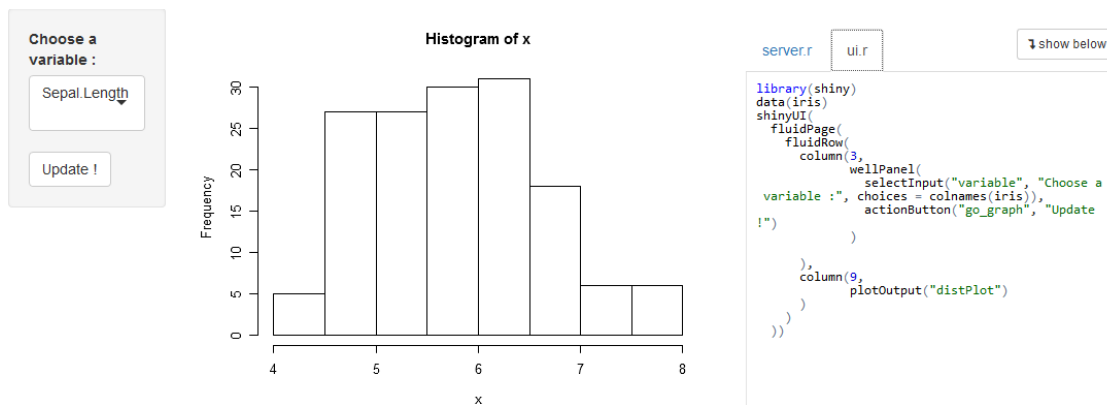
- L'option `options(shiny.error = browser)` permet de lancer un `browser()` automatiquement lors de l'apparition d'une erreur :

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

13.4 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 :

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

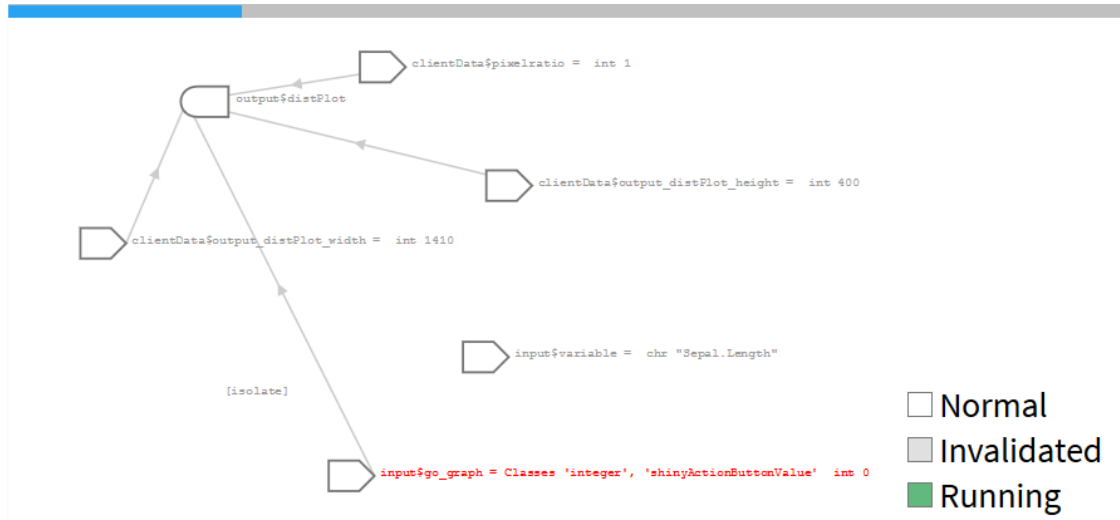


13.5 Reactive log

- 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)
})
```

13.6 Communication client/server

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

`options(shiny.trace = TRUE)`

```

Console | R Markdown
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"}}

```

13.7 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 différentes fonctions internes, avec `options(shiny.fullstacktrace = TRUE)`.

`options(shiny.fullstacktrace = TRUE)`

```

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: ...

```

14 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* (<https://rstudio.github.io/packrat/>).
- Mettre toute la **partie "calcul"** dans des **fonctions/un package** et effectuer des tests (cf. <http://r-pkgs.had.co.nz/tests.html>).
- Diviser la partie **ui.R** et **server.R** en plusieurs scripts, un par onglet par exemple :

```

# 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)
})

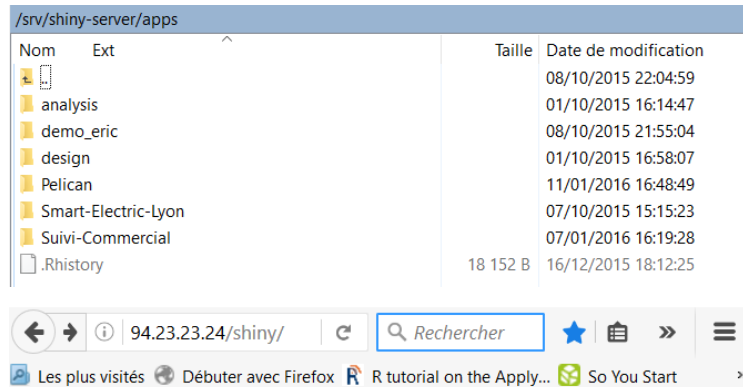
```

15 Quelques mots sur shiny-server

On peut déployer en interne nos applications shiny en installant un shiny-server (<https://www.rstudio.com/products/shiny/shiny-server2/>).

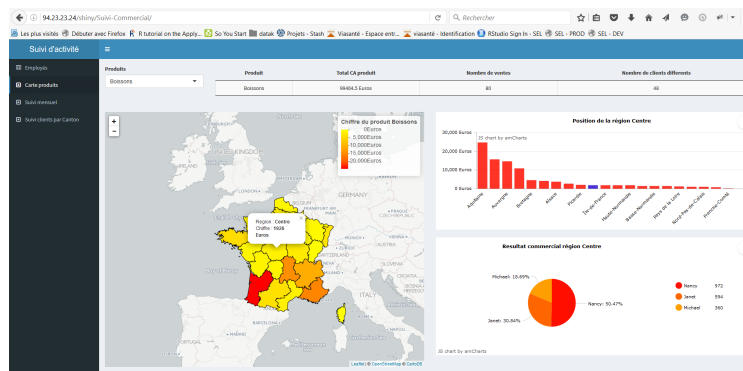
- Uniquement sur linux : ubuntu 12.04+, RedHat/CentOS 5+, SUSE Enterprise Linux 11+
- Version gratuite : déployer plusieurs applications **shiny**
- Version payante :
 - authentification ;
 - ressources par applications (nombre de coeurs, mémoire, ...) ;
 - monitoring.

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 ».

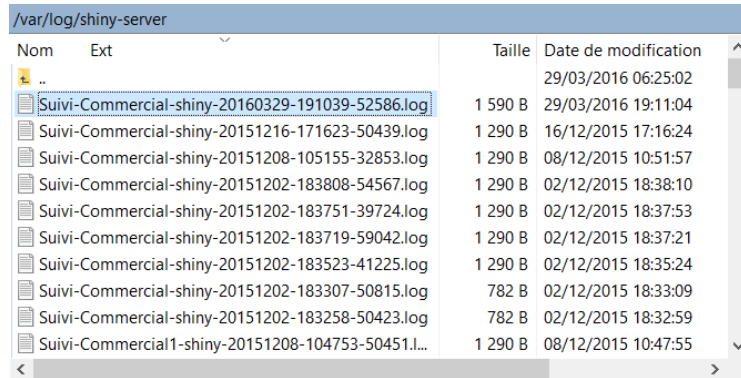


Index of /apps/

- [analysis/](#)
- [demo_eric/](#)
- [design/](#)
- [Pelican/](#)
- [Smart-Electric-Lyon/](#)
- [Suivi-Commercial/](#)



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



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

15.1 Références / Tutoriaux / Exemples

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