Premiers pas avec shiny… et les packages de visualisation interactive

B.Thieurmel, [benoit.thieurmel@datastorm.fr](mailto:benoit.thieurmel@datastorm.fr)

04/07/2018

# Shiny : créer des applications web avec le logiciel R

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

* pas de connaissances *web* nécessaires
* le pouvoir de calcul de R et l’intéractivité du web actuel
* pour créer des applications locales
* ou partagées avec l’utilisation de **shiny-server**, **shinyapps.io**, **shinyproxy**

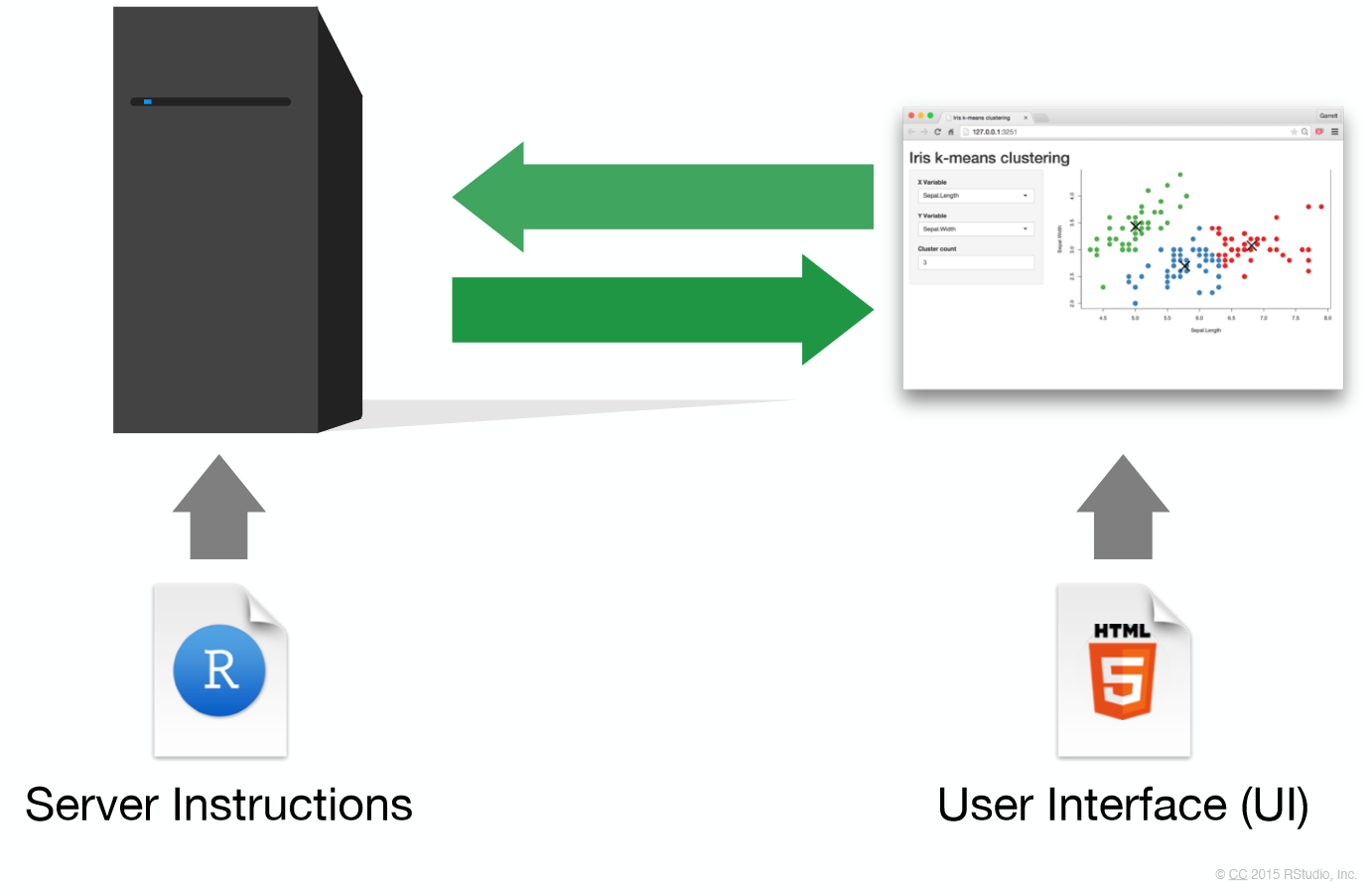
<http://shiny.rstudio.com>

<http://www.shinyapps.io/>

<https://www.shinyproxy.io/>

<https://www.rstudio.com/products/shiny/shiny-server/>

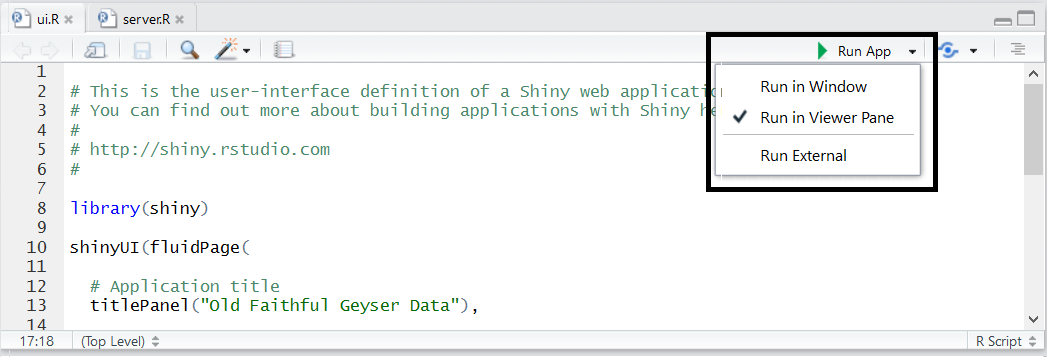
Une application **shiny** nécessite un ordinateur/un serveur éxécutant **R**



server

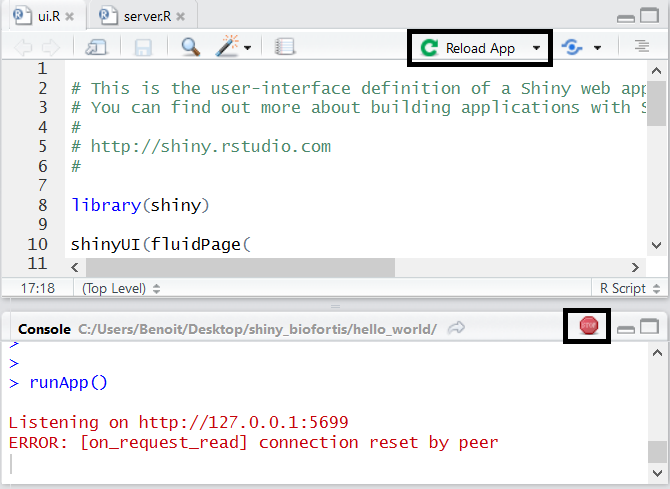
# Ma première application avec shiny

* 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**
* actualisatisation : bouton **Reload app**
* arrêt : bouton **Stop**



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



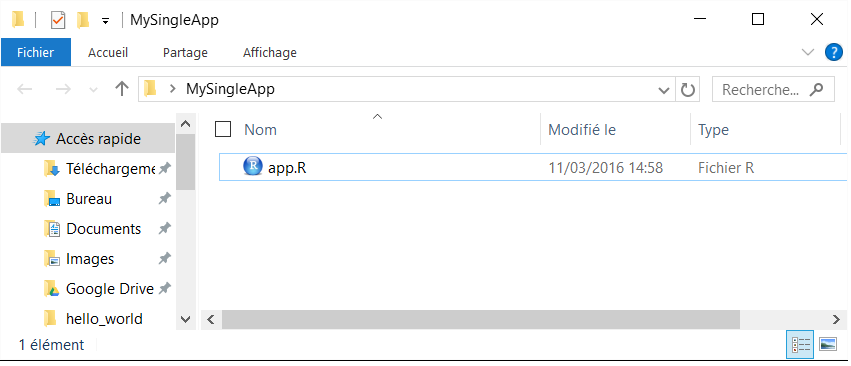
stop

# Structure d’une application

## Un dossier avec un seul fichier

**conventions :**

* enregistré sous le nom **app.R**
* se terminant par la commande shinyApp()
* pour les **applications légères**



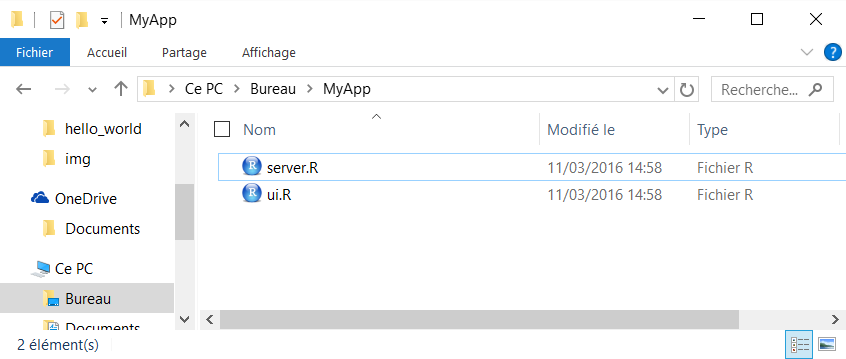
Understand communication

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)

## Un dossier avec deux fichiers

**conventions :**

* côté interface utilisateur dans le script **ui.R**
* côté serveur dans le script **server.R**
* structure à **priviliégier**



Understand communication

**ui.R**

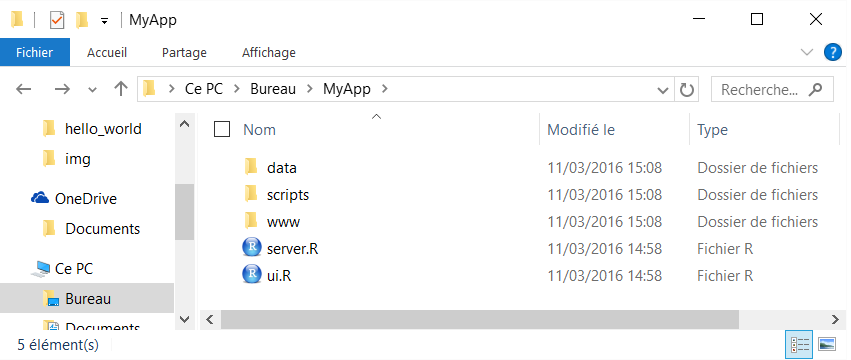
library(shiny)  
fluidPage(  
 sliderInput(inputId = "num", label = "Choose a number",   
 value = 25, min = 1, max = 100),   
 plotOutput("hist")  
)

**server.R**

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

## 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 (css, images, javascript, documentation, …)



Understand communication

## 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 seul 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

# Intéractivité et communication

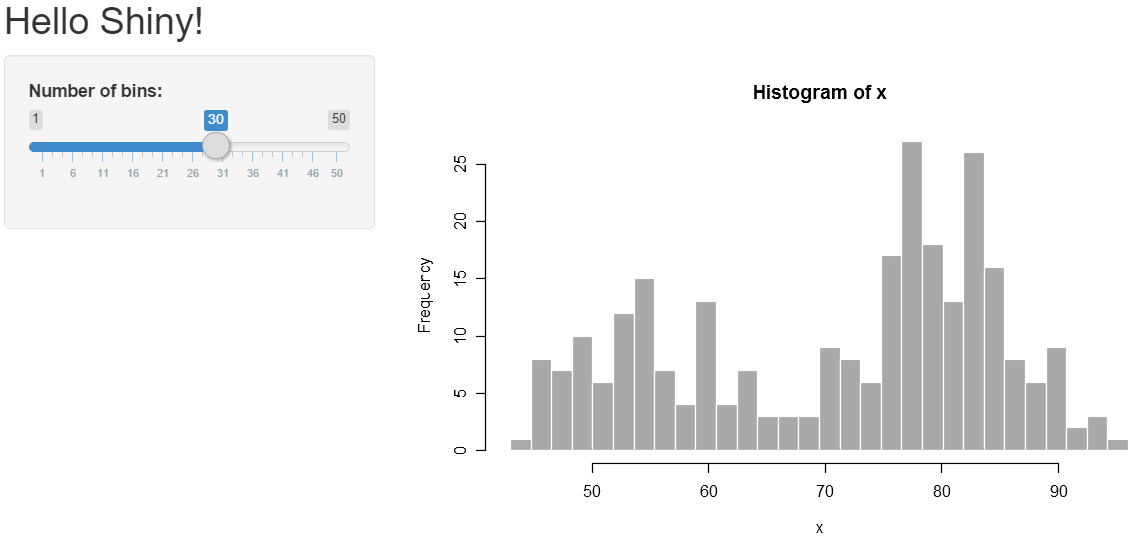
## Introduction

**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  
 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')  
 })  
})

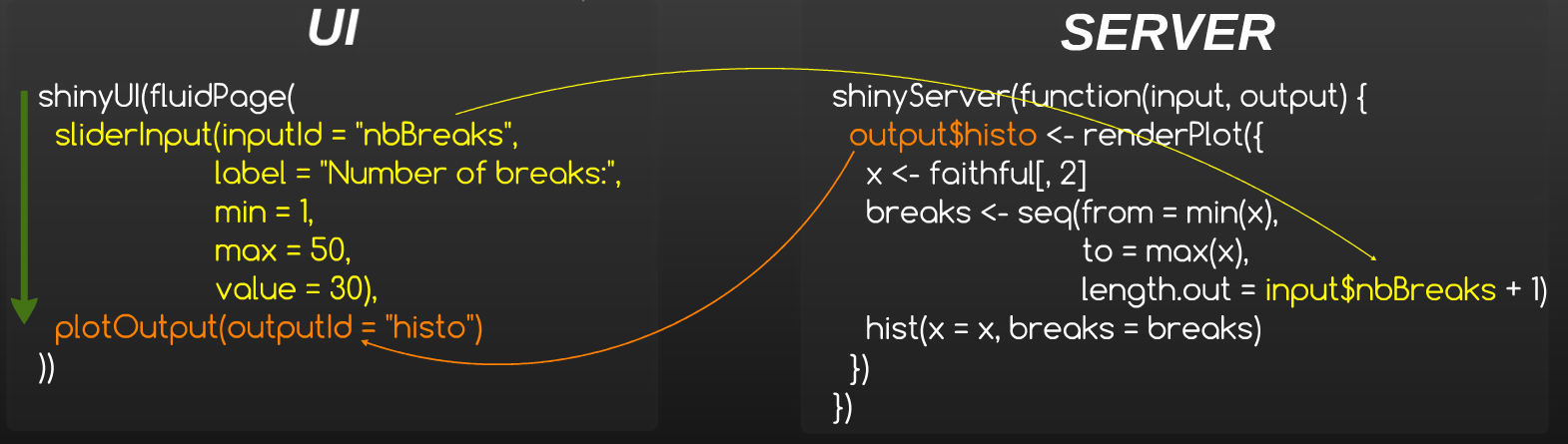


hello

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$bins” : c’est comme cela que le **ui** créé des variables disponibles dans le **server** !
* Côété **server**, nous créons un graphique “output$distPlot <- renderPlot({...})” et l’appelons dans le **ui** avec “plotOutput(outputId = "distPlot")”, c’est comme cela que le **server** retourne des objet à **ui** !

## Process



Understand communication

**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**

## Notice

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

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

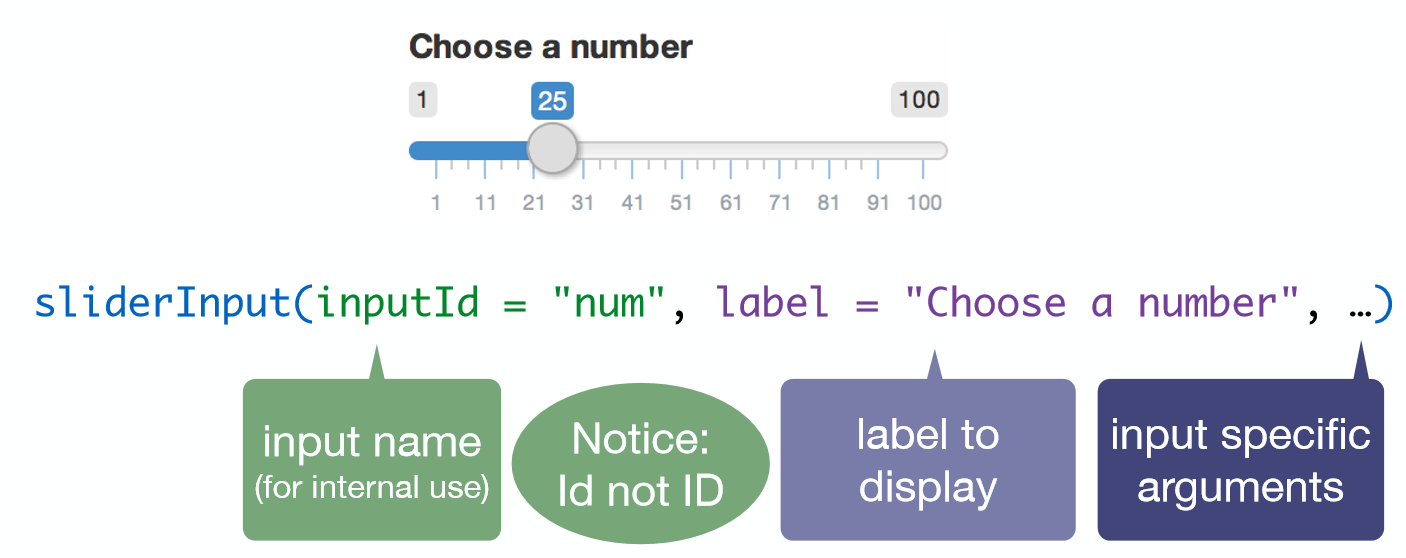
**la partie serveur/calculs : SERVER**

* la déclaration et le calcul des outputs

## UI

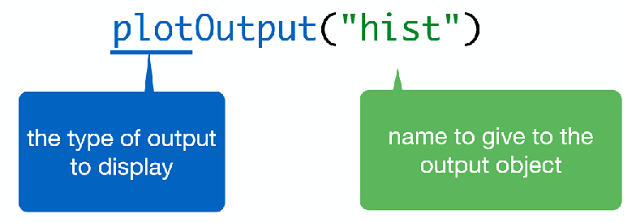
**Deux types d’éléments dans le UI**

* xxInput(inputId = …, …):
* définit un élément qui permet une action de l’utilisateur
* accessible côté serveur avec son identifiant **input$inputID**



Understand communication

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

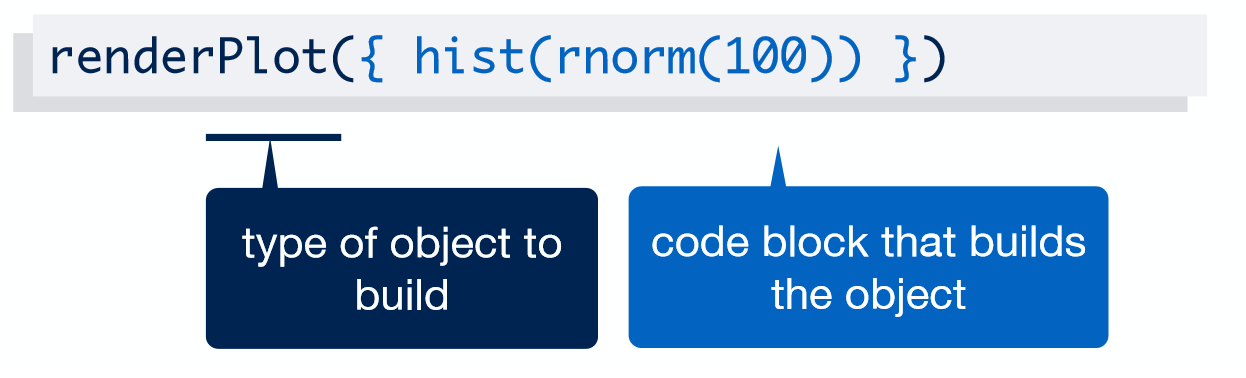


Understand communication

## Serveur

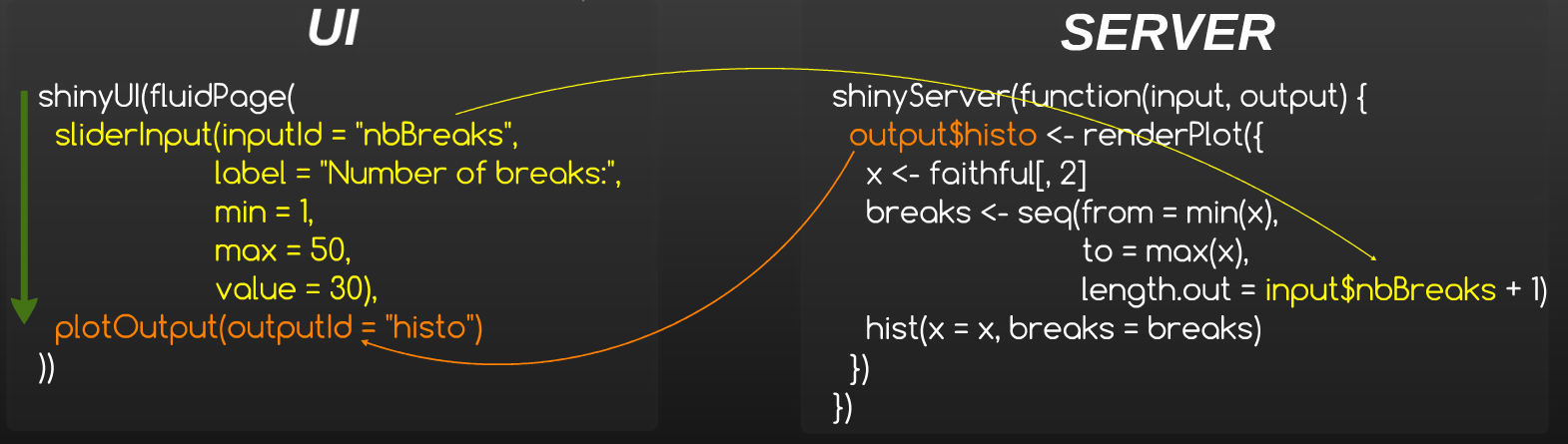
**Définition des outputs dans le serveur**

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



Understand communication

## Retour sur le process

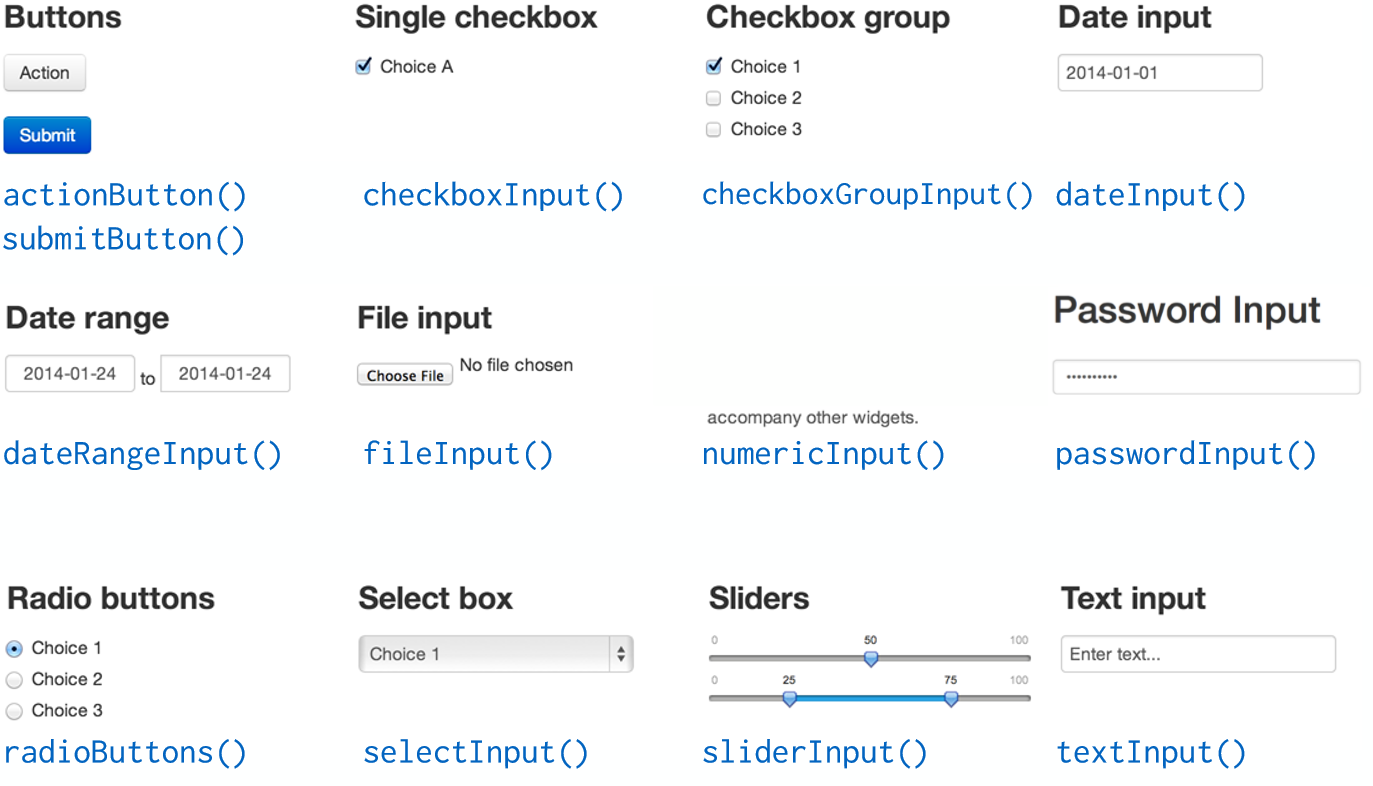


Understand communication

**C’est plus clair ?**

# Les inputs

## Vue globale



Understand communication

## 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)



numeric

## 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"



text

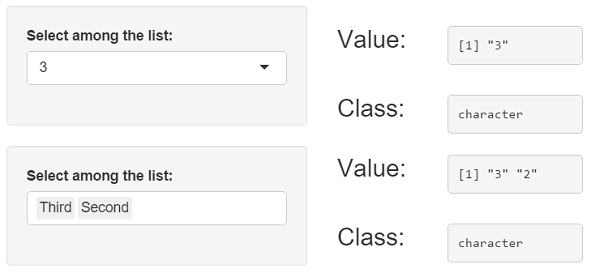
## Liste de sélection

* 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)



us

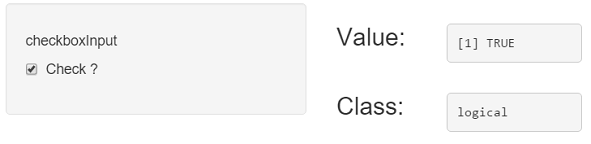
## Checkbox

* La fonction

checkboxInput(inputId, label, value = FALSE)

* Exemple:

checkboxInput(inputId = "idCheck1", label = "Check ?")  
  
# For the server input$idCheck1 is of class "logical"



sc

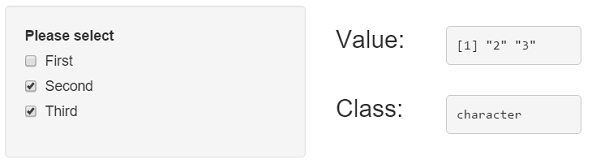
## Checkboxes multiple

* 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



mc

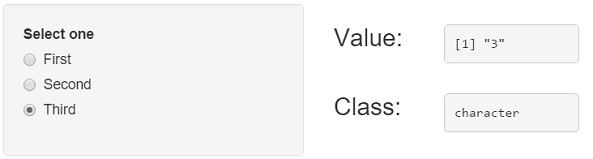
## Radio boutons

* 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"



rb

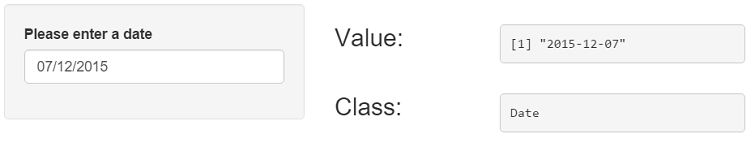
## Date

* 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"



d

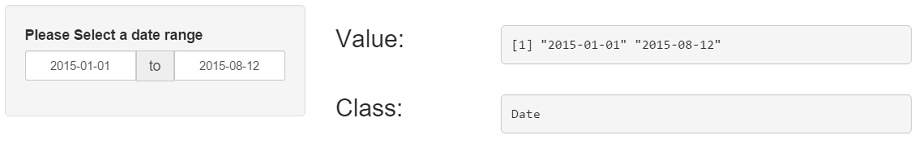
## Période

* 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



dr

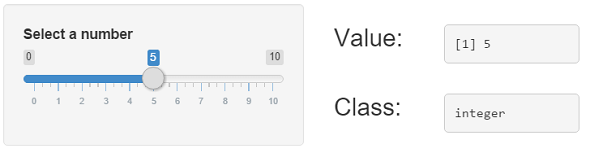
## Slider numérique : valeur unique

* 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)



sl

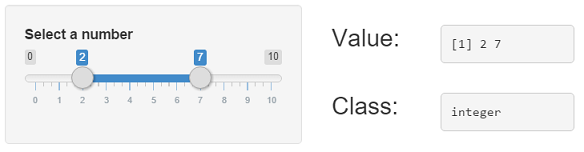
## Slider numérique : range

* 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)



msl

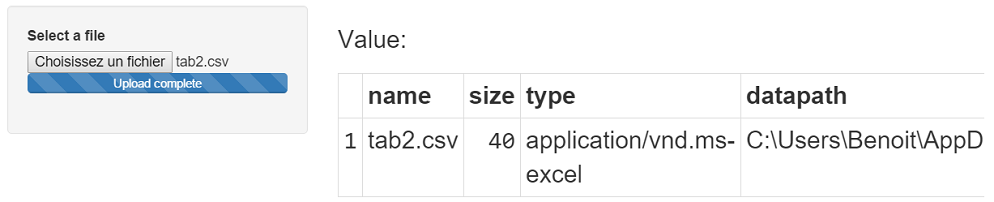
## Importer un fichier

* 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



up

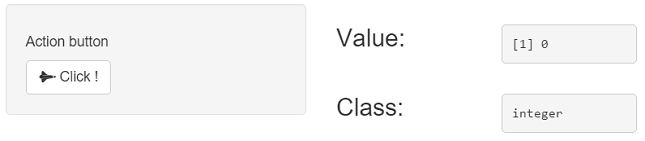
## Action Bouton

* 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"



ab

## 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**

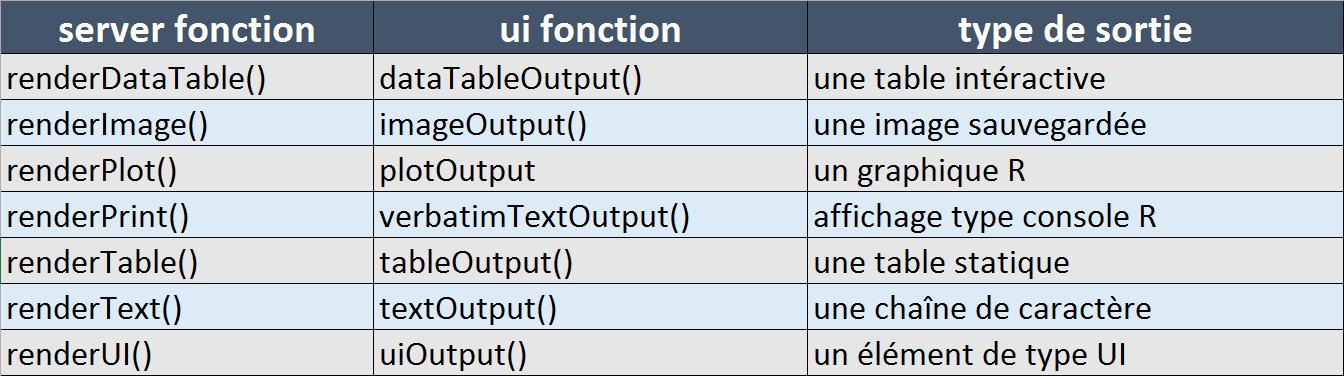
Un tutoriel est disponible : <http://shiny.rstudio.com/articles/building-inputs.html>

Ainsi que deux applications d’exemples :

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

# Outputs

## Vue globale



ab

## 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})

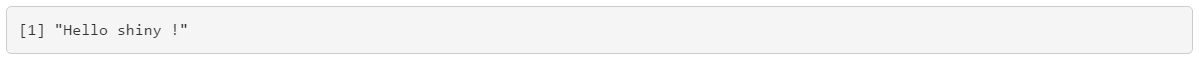
## Print

* **ui.r**:

verbatimTextOutput(outputId = "texte")

* **server.r**:

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



op

## Text

* **ui.r**:

textOutput(outputId = "texte")

* **server.r**:

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

op2

op2

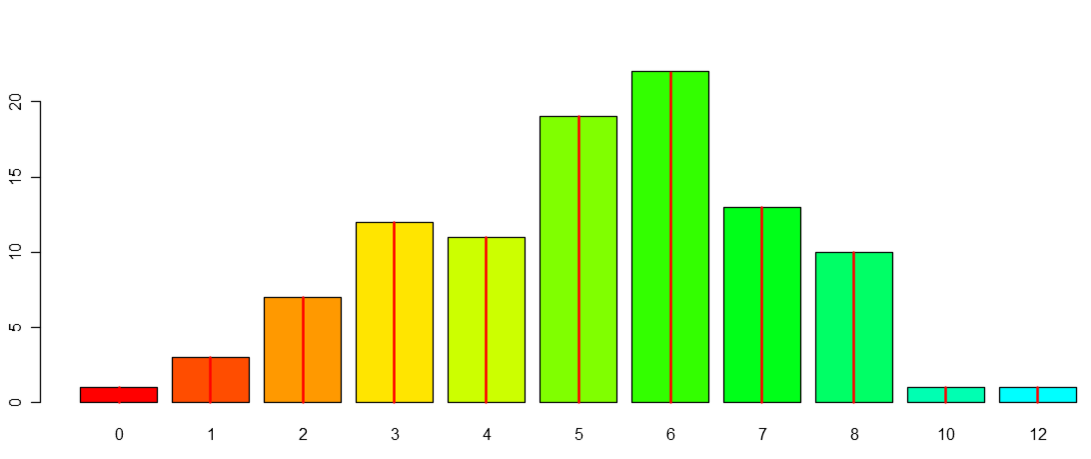
### Plot

* **ui.r**:

plotOutput("myplot")

* **server.r**:

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



op3

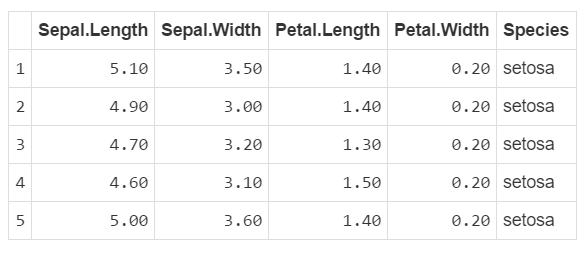
## Table

* **ui.r**:

tableOutput(outputId = "table")

* **server.r**:

output$table <- renderTable({iris})



op4

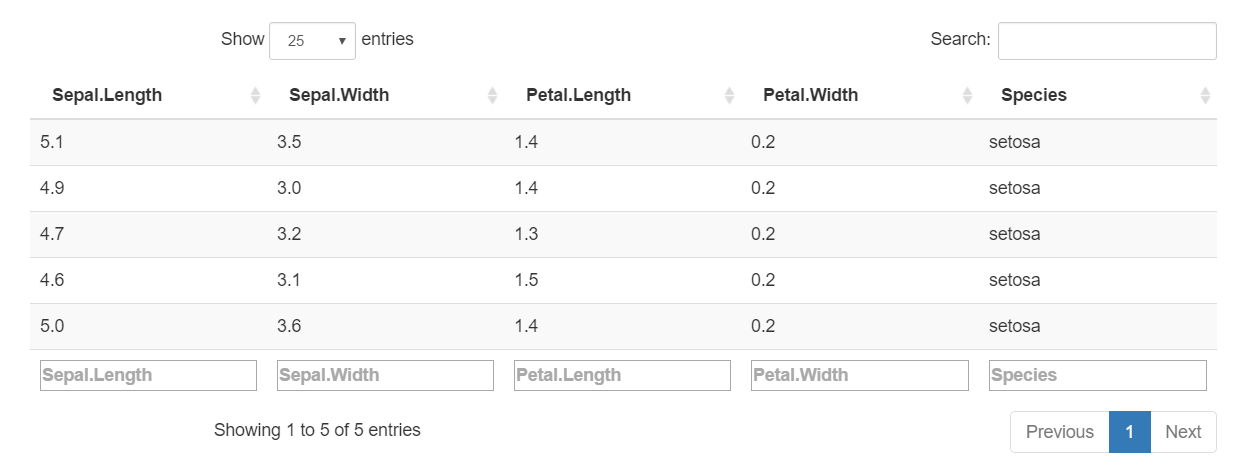
## DataTable

* **ui.r**:

dataTableOutput(outputId = "dataTable")

* **server.r**:

output$dataTable <- renderDataTable({  
 iris  
})



op5

## Définir des élements de l’UI côté SERVER | Définition

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

Cela est possible avec les fonctions uiOutput et renderUI

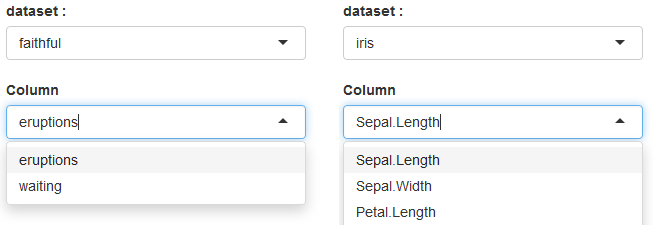
## Définir des élements de l’UI côté SERVER | Exemple simple

* **ui.r**:

uiOutput(outputId = "columns")

* **server.r**:

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



op5

## Définir des élements 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")))  
 )  
)

## 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.

# Structurer sa page

## 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")  
 )  
))

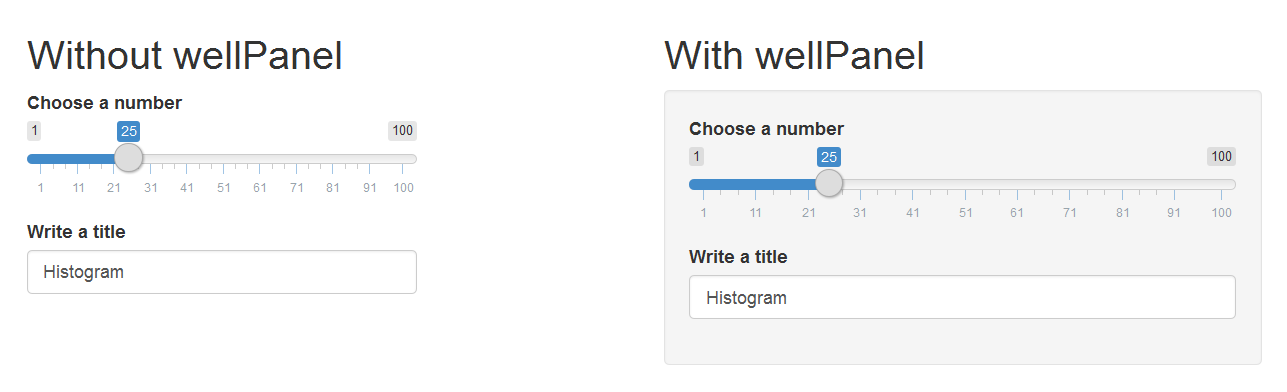


sdb

## 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")  
))



sdb

## 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")  
 )  
 )  
)

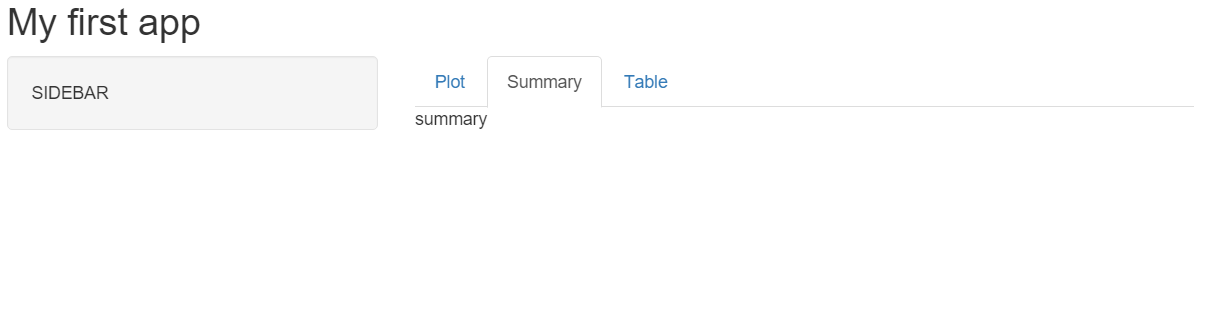


nav

## 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"))  
 )  
 )  
 )  
))



pan

## 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"))  
 )  
))



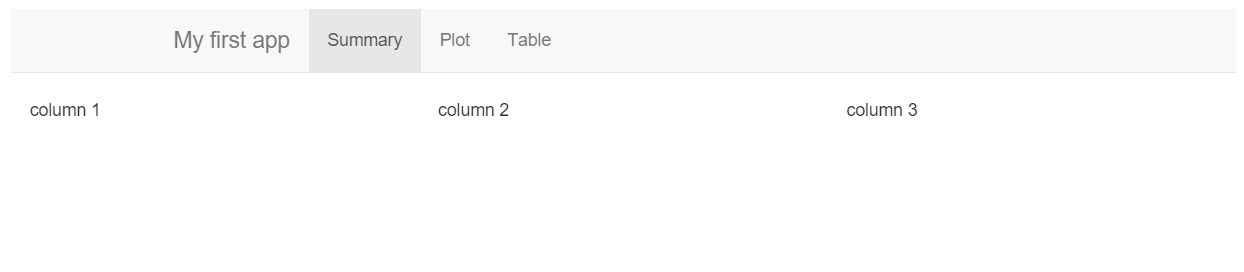
pan

## 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

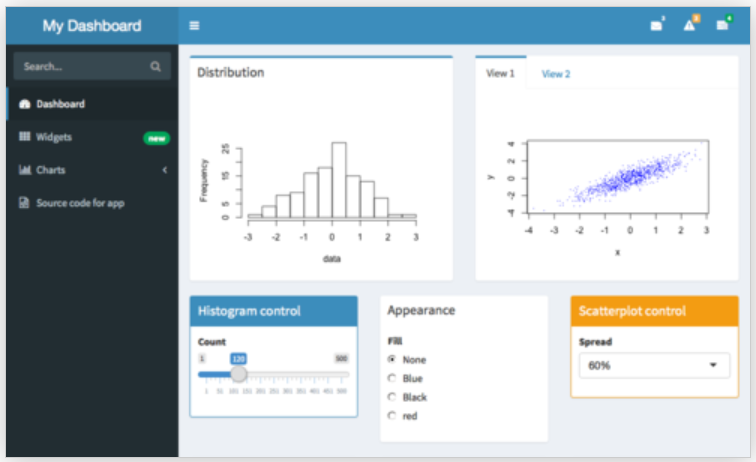
tabPanel(title = "Summary",  
 # A fluid row can contain from 0 to 12 columns  
 fluidRow(  
 # A column is defined necessarily  
 # with its argument "width"  
 column(width = 4, "column 1"),  
 column(width = 4, "column 2"),  
 column(width = 4, "column 3"),  
 ))



grid

## shinydashboard

Le package [shinydashboard](https://rstudio.github.io/shinydashboard/) propose d’autres fonctions pour créer des tableaux de bords :

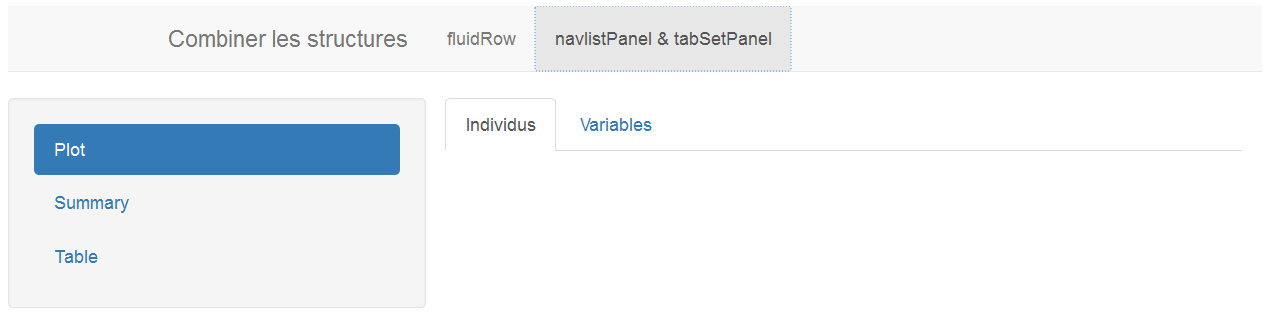


grid

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

## Combiner les structures

Toutes les structures peuvent s’utiliser en même temps !



grid

# Graphiques intéractifs

Avec notamment l’arrivée du package [htmlwidgets](http://www.htmlwidgets.org/), de plus en plus de fonctionnalités de librairies javascript sont accessibles sous **R** :

* [dygraphs (time series)](http://rstudio.github.io/dygraphs/)
* [DT (interactive tables)](http://rstudio.github.io/DT/)
* [Leafet (maps)](http://rstudio.github.io/leaflet/)
* [d3heatmap](https://github.com/rstudio/d3heatmap)
* [threejs (3d scatter & globe)](http://bwlewis.github.io/rthreejs)
* [rAmCharts](http://datastorm-open.github.io/introduction_ramcharts/)
* [visNetwork](http://datastorm-open.github.io/visNetwork)
* …

Plus généralement, jeter un oeil sur la [gallerie suivante!](http://gallery.htmlwidgets.org/)

## 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](http://rstudio.github.io/dygraphs/) :

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

Ces packages arrivent souvent avec des méthodes permettant d’intéragir avec le graphique, en créant des inputs dans **shiny** afin de déclencher des actions . Par exemple :

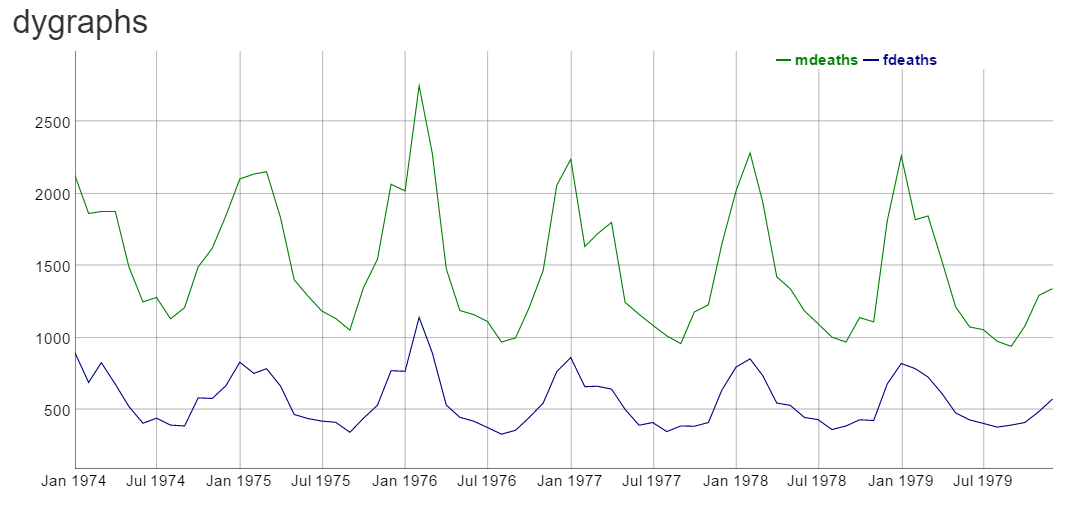
* **DT** : création de *input$tableId\_rows\_selected*, nous informant sur la/les lignes sélectionnée(s)
* **Leaflet** : valeurs du zoom, des clicks, de la latitude/longitude, …
* **visNetwork** : noeuds / groupes sélectionnés, …

Ces points sont (en général) expliqués sur les pages web des différents packages…

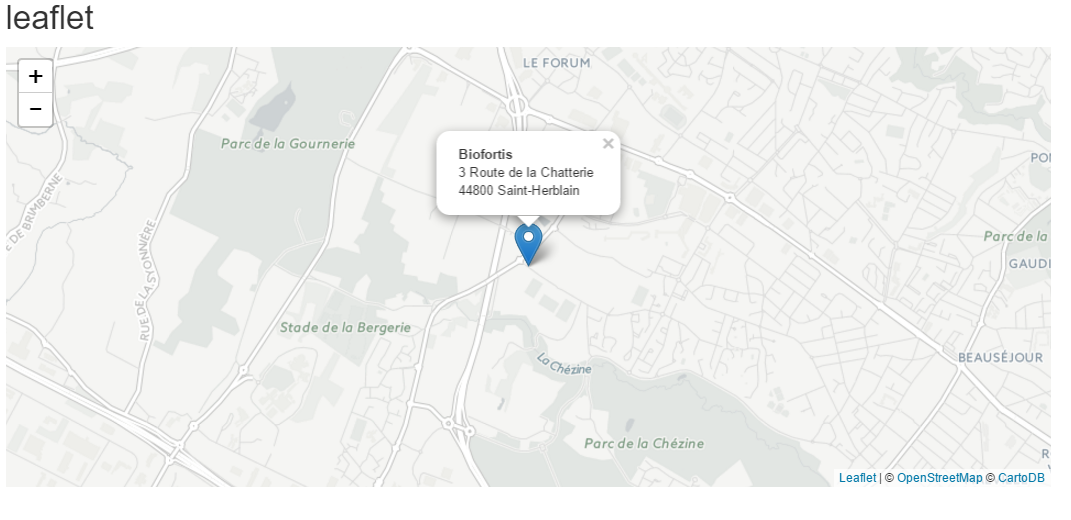
De plus, il est également possible d’utiliser de nombreux événements javascripts, et de crééer des nouvelles intéractions avec **shiny** en utilisant *Shiny.onInputChange* :

visNetwork(nodes, edges) %>%  
 visEvents(hoverNode = "function(nodes) {  
 Shiny.onInputChange('current\_node\_id', nodes);  
 ;}")

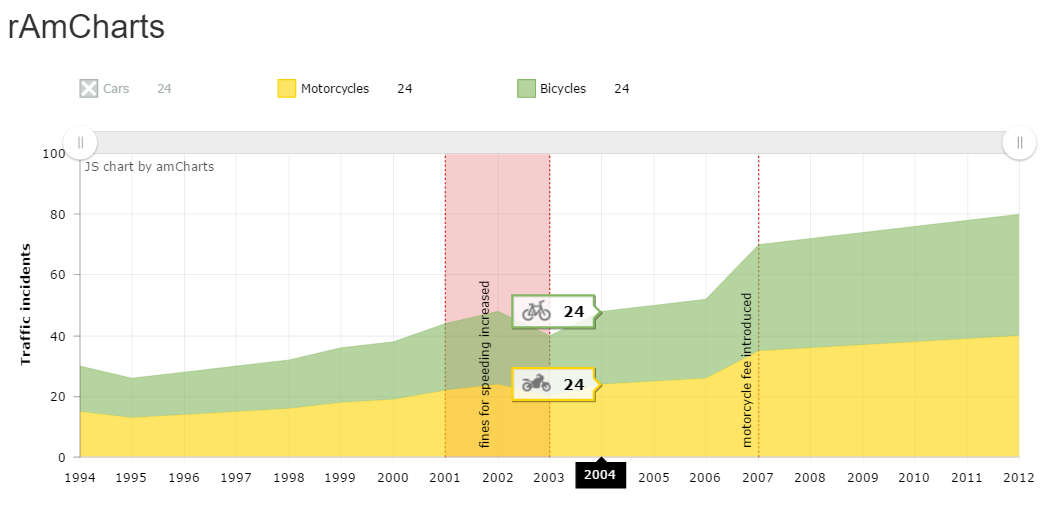
<https://shiny.rstudio.com/articles/js-send-message.html>



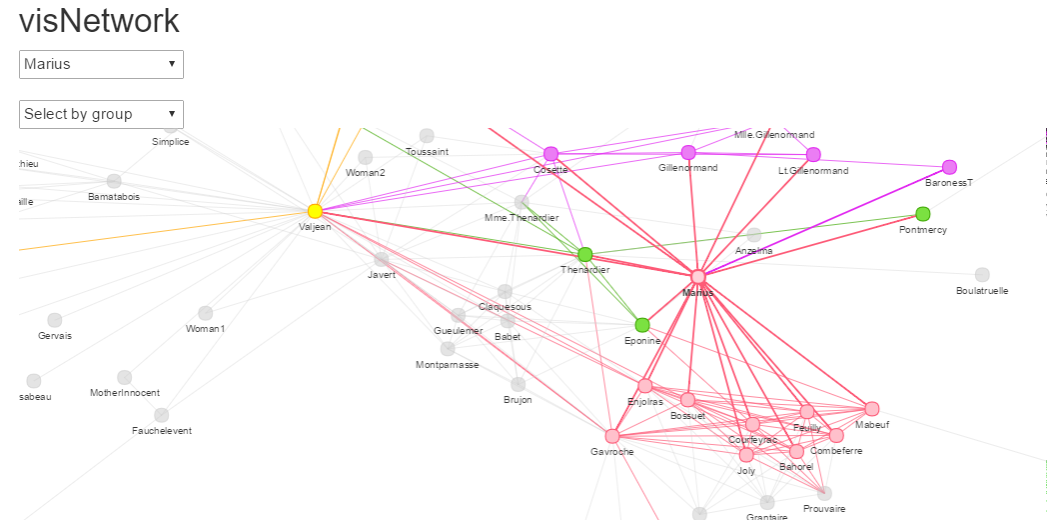
dyg



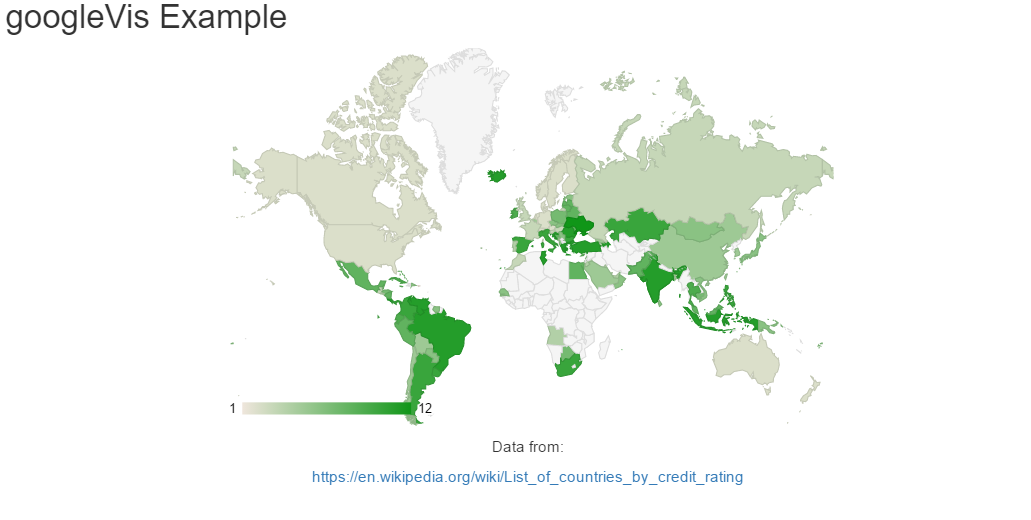
leaf



ram



vis



ggv

# Isolation

## 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 {}

## Exemple 1

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

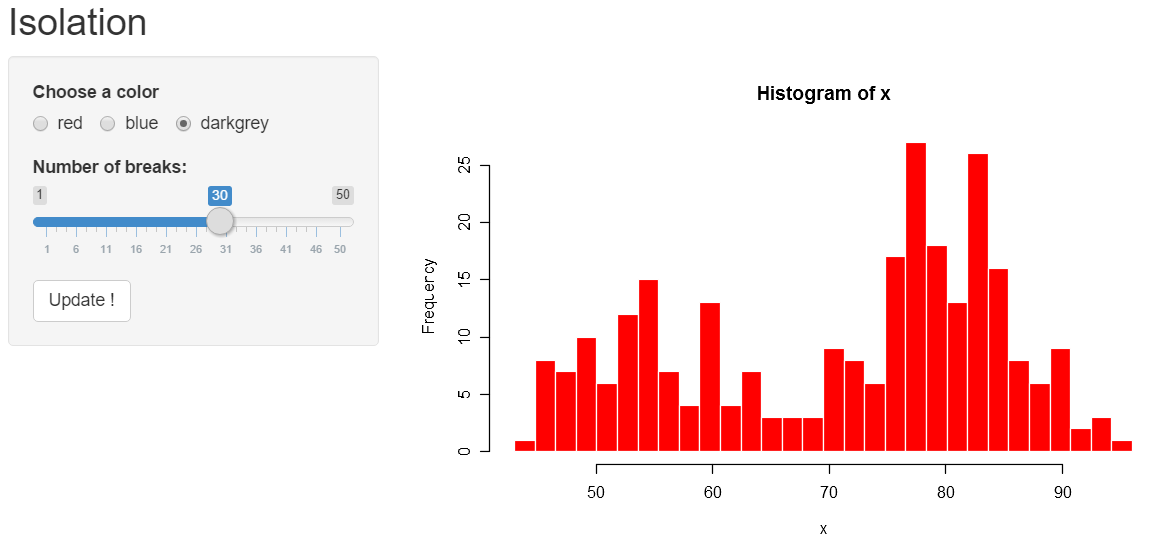
shinyUI(fluidPage(  
 titlePanel("Isolation"),  
 sidebarLayout(  
 sidebarPanel(  
 radioButtons(inputId = "col", label = "Choose a color", inline = TRUE,  
 choices = c("red", "blue", "darkgrey")),  
 sliderInput("bins", "Number of bins:", min = 1, max = 50, value = 30),  
 actionButton("go\_graph", "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.



iso

## 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.

# 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

## 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))  
})

## 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))  
})

## Note

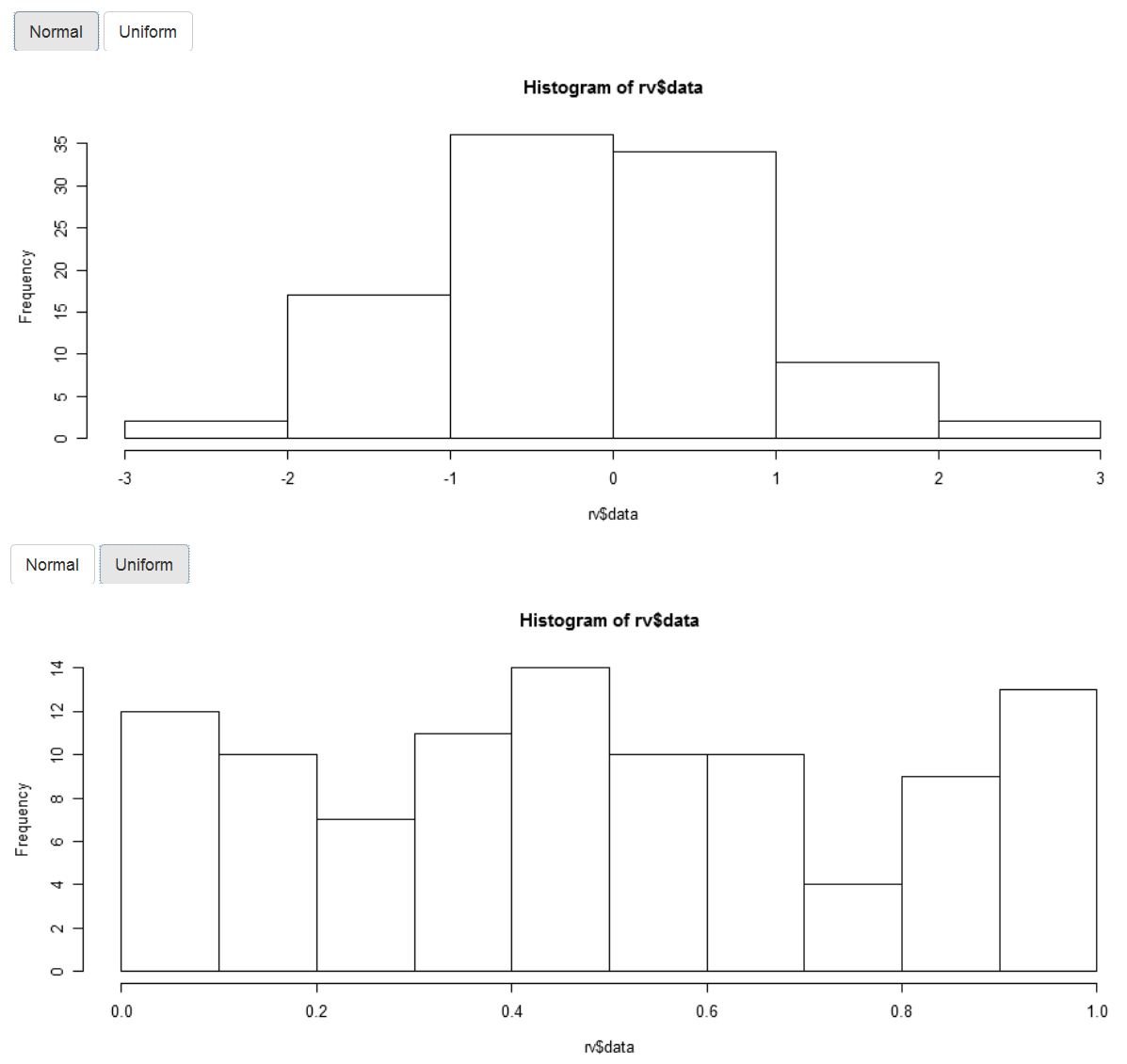
* Une expression réactive va nous faire gagner du temps et de la mémoire
* **Utiliser des expressions réactives seulement quand cela dépend d’inputs** (pour d’autres variables : <http://shiny.rstudio.com/articles/scoping.html>)
* **Comme un output** : mis-à-jour chaque fois qu’un input présent dans le 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 “()”.

## Autres fonctions

Il existe des alternatives à l’utilisation de reactive avec reactiveValues ou reactiveVal.

* reactiveValues : initialiser une liste d’objets réactifs
* reactiveVal : initialiser un seul objet réactif
* Modification de la valeur des objets avec des observe ou des observeEvent

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) })   
})



# Conditional panels

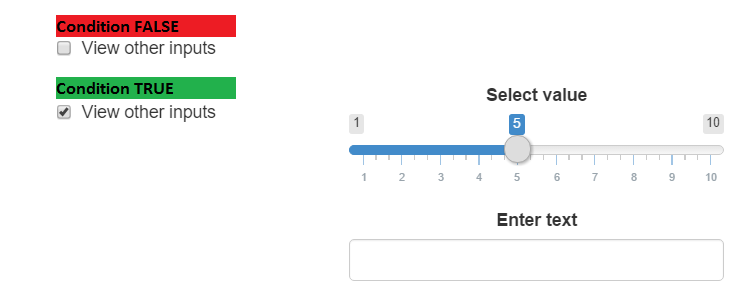
* Il est possible d’afficher conditionnellement ou non certains éléments :

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) {}  
)



cond1

# Observe & fonctions d’update

## 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**

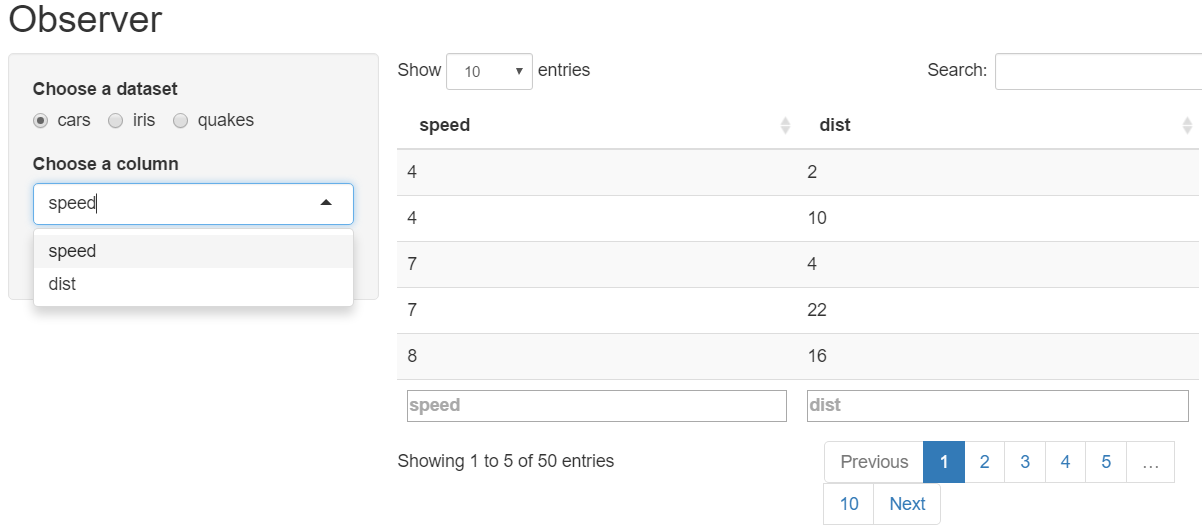
Pour changer dynamiquement l’onglet sélectionné :

* **updateNavbarPage**, **updateNavlistPanel**, **updateTabsetPanel**

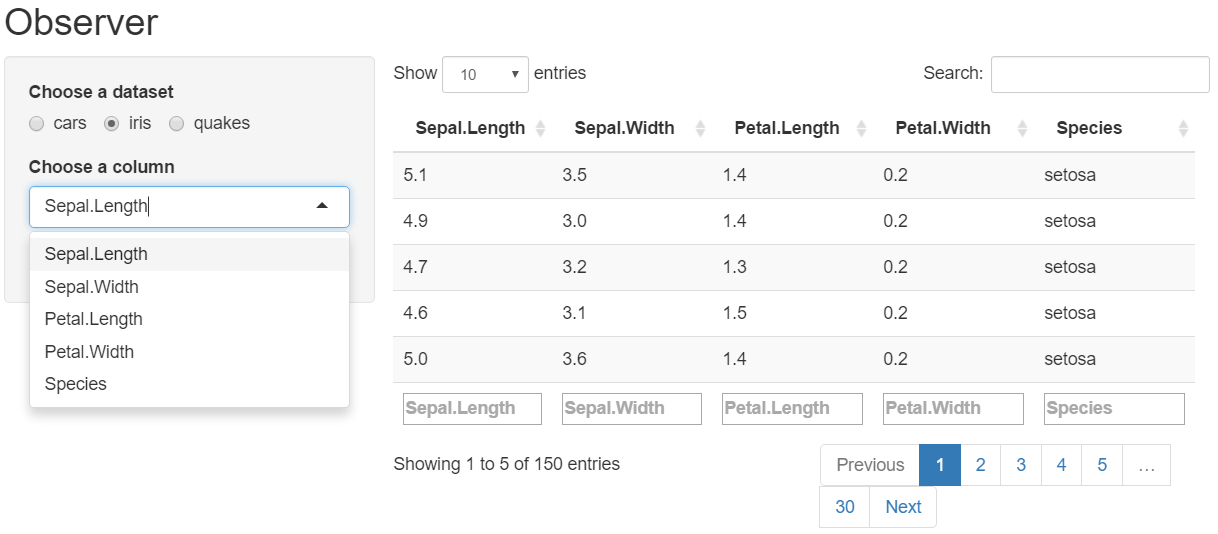
## Exemple sur un input

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)  
 )  
})



obs1



obs2

## Exemple sur des onglets

**Il faut rajouter un id dans la structure**

shinyUI(  
 navbarPage(  
 id = "idnavbar", # need an id for oberve & 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")  
 })  
})

## observeEvent

* Une variante de la fonction observe est disponible avec la fonction observeEvent
* On définit alors de façon explicite l’espression qui représente l’événement *et* l’expression qui sera éxécutée 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")  
})

# HTML / CSS

## 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"



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>")   
)

## Quelques balises utiles

* div(..., align = "center") : centrer les éléments
* br() : saut de ligne
* hr() : trait horizontal
* img(src="img/logo.jpg", title="Popup", width = "80%") : insertion d’une image présente dans **www/img**
* a(href="https://r2018-rennes.sciencesconf.org/", target="\_blank", "Rencontres R") : lien vers un site
* a(href = './doc/guide.pdf', target="\_blank", class = "btn", icon("download"), 'Télécharger le guide utilisateur') : lien de téléchargement d’un document présent dans **www/doc**

## CSS : introduction

**Shiny** utilise [Bootstrap](http://getbootstrap.com/) 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

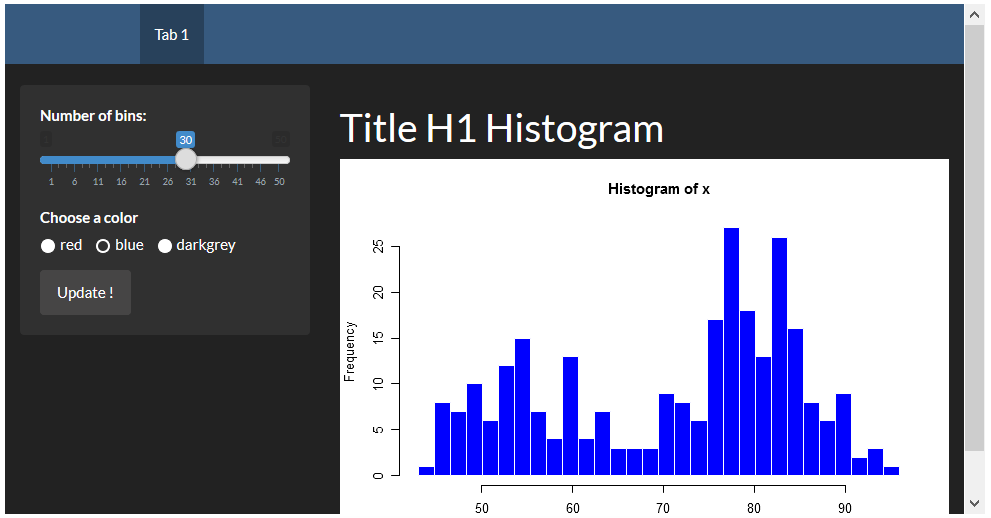
On peut aussi utiliser le package [shinythemes](http://rstudio.github.io/shinythemes)

## Avec un .css externe

On peut par exemple aller prendre un thème sur [bootswatch](http://bootswatch.com/).

* 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  
)

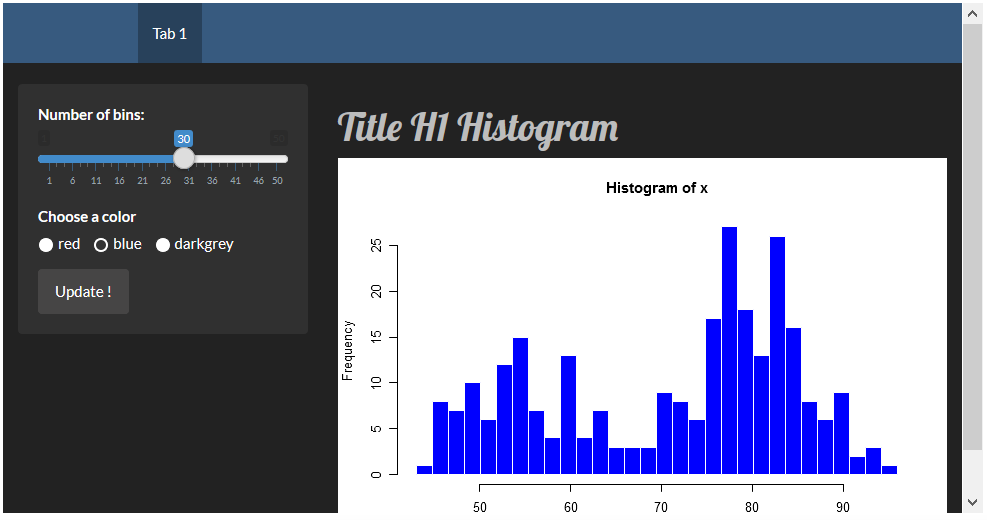


css1

## 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  
)

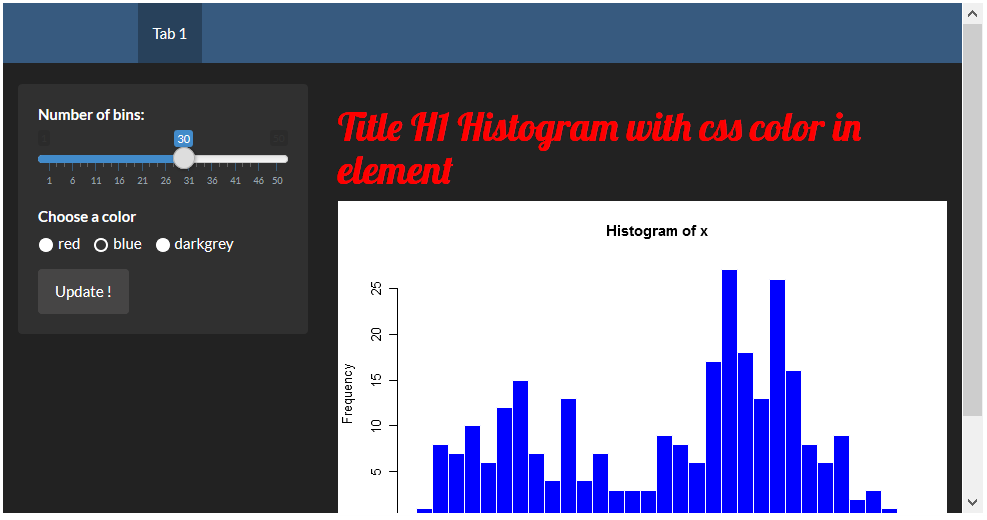


css2

## CSS sur un élément

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

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



css3

# Quelques bonnes pratiques

* Préférer l’underscore (\_) au point (.) comme séparateur dans le nom des variables. En effet, le **.** peut amener de mauvaises intérations 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 ([testthat](http://r-pkgs.had.co.nz/tests.html))
* Diviser la partie **ui.R** et **server.R** en plusieurs scripts, un par onglet par exemple :

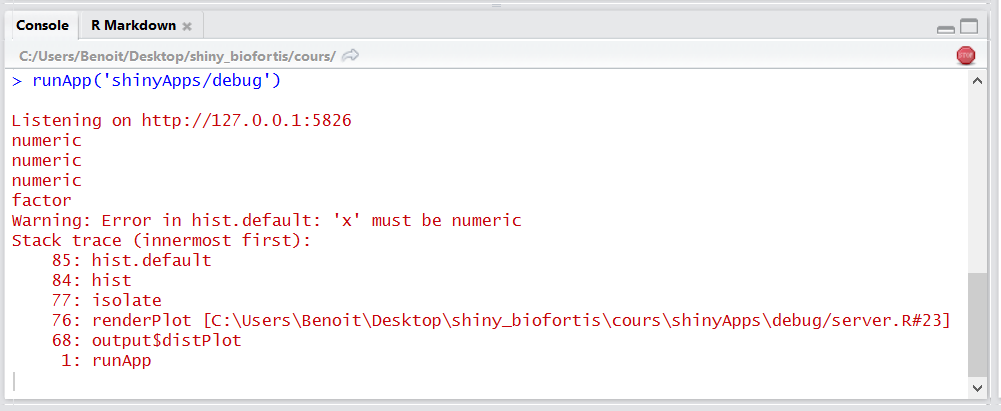
# ui.R  
shinyUI(  
 navbarPage("Divide UI & SERVER",  
 source("src/ui/01\_ui\_plot.R", local = TRUE)$value,  
 source("src/ui/02\_ui\_data.R", local = TRUE)$value  
 )  
)  
# server.R  
shinyServer(function(input, output, session) {  
 source("src/server/01\_server\_plot.R", local = TRUE)  
 source("src/server/02\_server\_data.R", local = TRUE)  
})

# Débogage

## 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’éxécution 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)  
})



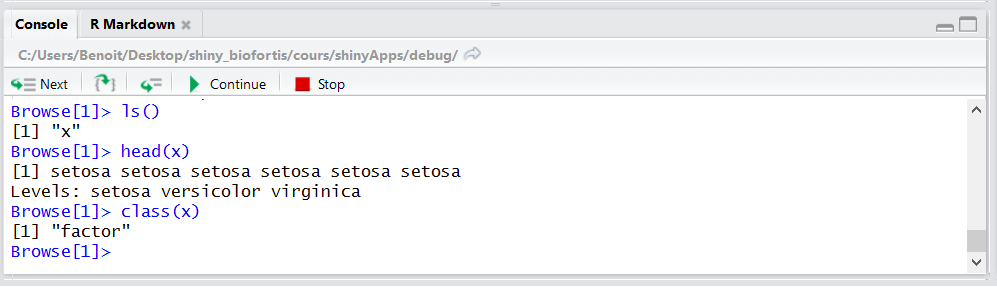
ggv

## 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…!



ggv

## Lancement automatique d’un browser

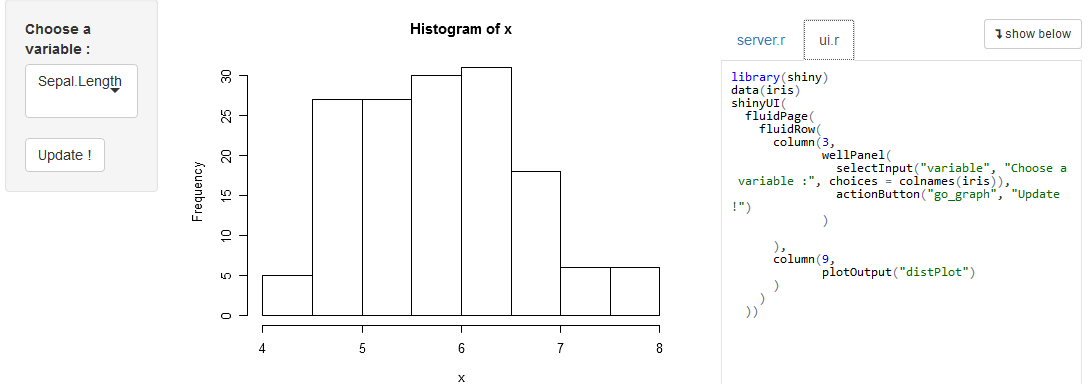
* L’option options(shiny.error = browser) permet de lancer un broswer() automatiquement lors de l’apparition d’une erreur

options(shiny.error = browser)

## 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’éxécution du code :

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



ggv

## 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 tappant 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)  
})

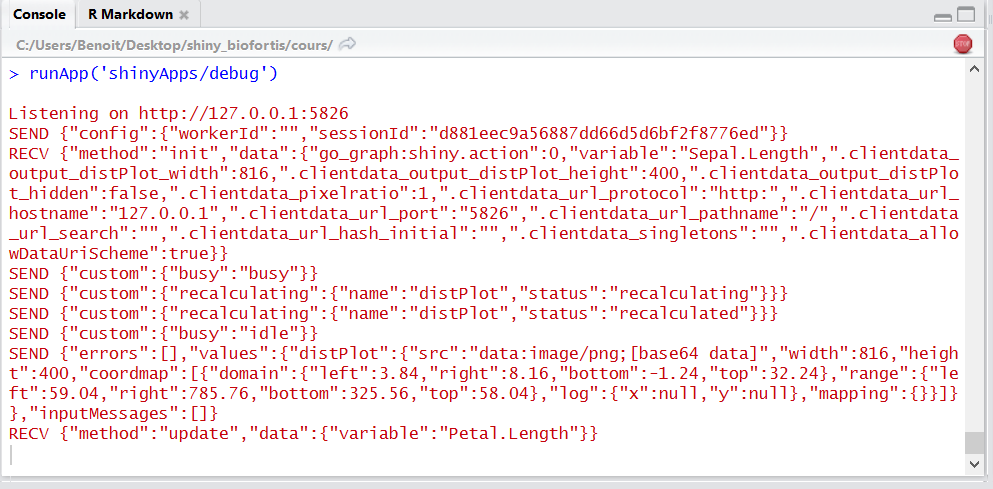


ggv

## Communication client/server

* Toutes les communications entre le client et le server sont visibles en utilisant l’option shiny.trace

options(shiny.trace = TRUE)

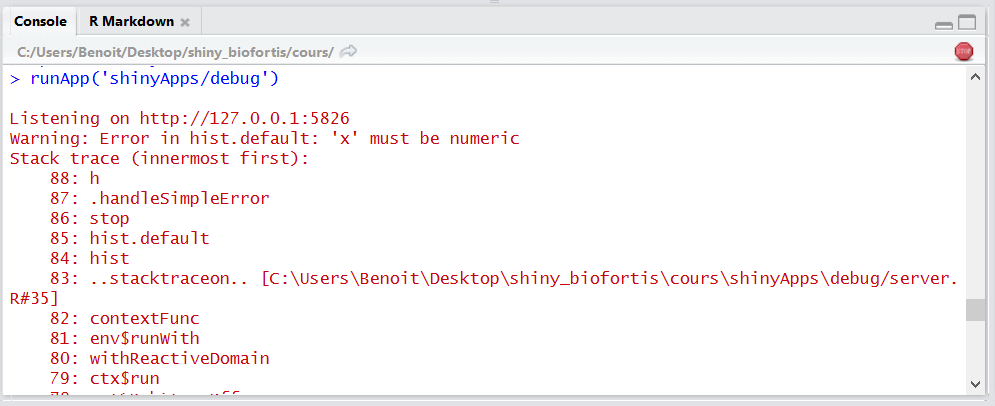


ggv

## 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 diffénrets fonctions internes, avec options(shiny.fullstacktrace = TRUE)

options(shiny.fullstacktrace = TRUE)



ggv

## 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/>