

Présentation du package Shiny

Éléments d'introduction au développement d'une application web réactive avec R

Benoit Thieurmél, benoit.thieurmél@datastorm.fr

Jeffery Petit, jeffery.petit@ls2n.fr

Dernière mäj en octobre 2018

Résumé

Ce tutoriel a été développé à des fins pédagogiques afin d'initier des développeurs au package shiny. Il s'agit d'un package riche permettant de créer des applications interactives reposant sur une session R. Cette introduction s'inspire de l'aide officielle (cf. <https://shiny.rstudio.com/tutorial>) et de notre expérience en programmation; elle est destinée à des personnes initiés à R qui n'ont pas nécessairement de connaissances en web. Associé à un support de présentation et des explications, nous espérons que cette présentation permettra au(x) lecteur(s) de prendre en main l'outil et de maîtriser les éléments de base pour être autonome. Nous faisons également mention de plusieurs fonctionnalités qui permettront aux intéressés d'approfondir leurs connaissances et d'améliorer la qualité de leurs applications.

Table des matières

1	Shiny : créer des applications web avec le logiciel R	4
2	Première application avec shiny	5
3	Les inputs	8
3.1	Valeurs numériques	9
3.1.1	Valeur au choix	9
3.1.2	Curseur : valeur unique	9
3.1.3	Curseur : intervalle	10
3.2	Les chaînes de caractère(s)	11
3.2.1	Valeur libre	11
3.2.2	Liste de sélection	11
3.3	Les cases à cocher	12
3.3.1	Une seule sélection	12
3.3.2	Sélection multiple	12
3.3.3	Choix alternatifs	13
3.4	Sélection de date(s)	13
3.4.1	Date unique	13
3.4.2	Période	14
3.5	Import d'un fichier	14
3.6	Le bouton de validation	15
4	Outputs	15
4.1	Textes	16
4.1.1	Verbatim	16
4.1.2	Texte standard	17
4.2	Graphiques	17
4.3	Tableaux	18
4.3.1	Tableau basique	18
4.3.2	Tableau interactif	18
4.4	Définir des éléments de l'UI côté SERVER	19
5	Arborescence d'une application	20
5.1	Codage de l'interface en HTML	22
5.2	Données/fichiers complémentaires	22
5.3	Partage ui <-> server	23
6	Structurer sa page	23

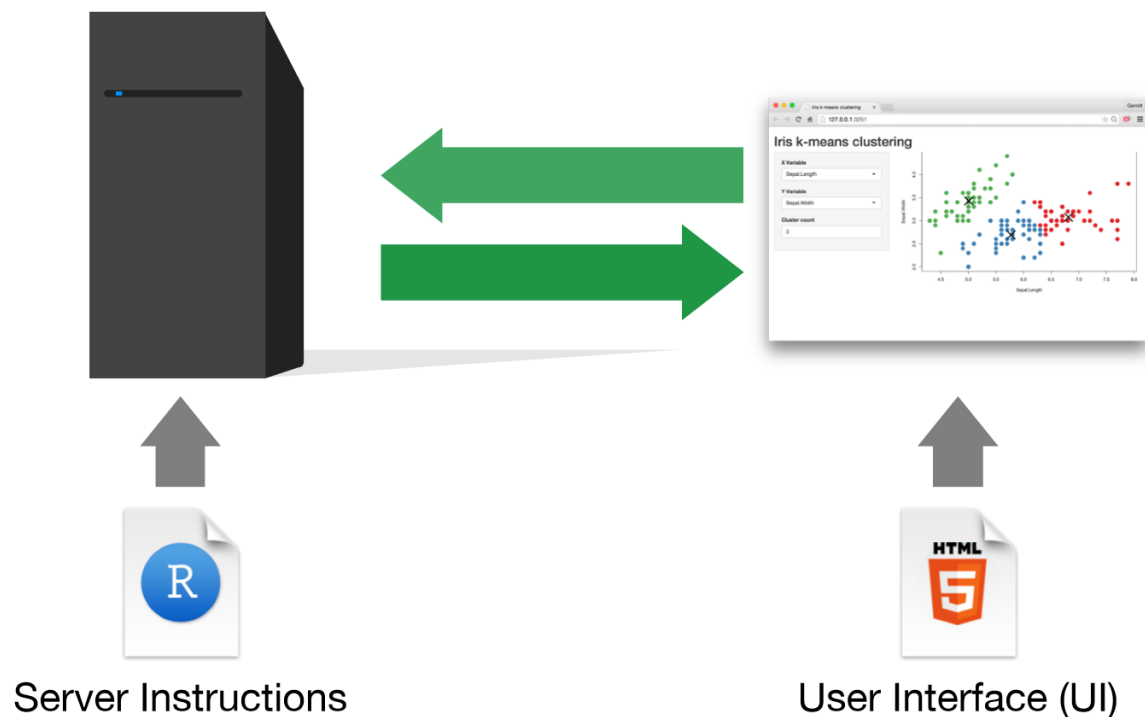
6.1	Division 1/3, 2/3	23
6.2	Le wellPanel	24
6.3	La navigation en onglets	24
6.4	Les onglet de sélection	25
6.5	Structure à la carte	27
6.6	Inclure du HTML	27
6.7	Un package, une interface : shinydashboard	29
7	Aspect général, personnalisation et réactivité	29
7.1	Customisation avec du CSS	29
7.2	Graphiques interactifs	32
8	Pour une meilleure maîtrise de la réactivité	34
8.1	Isolation	34
8.1.1	Isolation par expression	35
8.1.2	Isolation par input	36
8.2	Les expressions réactives	36
8.3	Les fonctions de mise à jour	38
8.4	Les élément d'interface conditionnels	42
9	Débogage	43
9.1	Affichage console	43
9.2	Lancement automatique d'un browser	44
9.3	Mode showcase	44
9.4	Reactive log	45
9.5	Communication client/server	45
9.6	Traçage des erreurs	46
10	Conclusion	47
10.1	Quelques bonnes pratiques	47
10.2	Quelques mots sur shiny-server	47
10.3	Références / Tutoriels / Exemples	49

1 Shiny : créer des applications web avec le logiciel R

Shiny [Chang et al., 2018]¹ est un package qui permet la création simple d'applications web interactives depuis le logiciel open-source **R** [R Core Team, 2018]. Il présente quelques intérêts non négligeables qui ont contribué à son succès et au soutien de la communauté :

- il ne nécessite pas de connaissances *web* ;
- les applications reposent sur la puissance de calcul **R** ;
- il permet de développer des applications interactives inspirées du web actuel ;
- il permet de créer des applications locales ou partagées avec l'utilisation d'un « *shiny-server* »².

Une application **shiny** nécessite un ordinateur/un serveur exécutant une session **R**.



© CC 2015 RStudio, Inc.

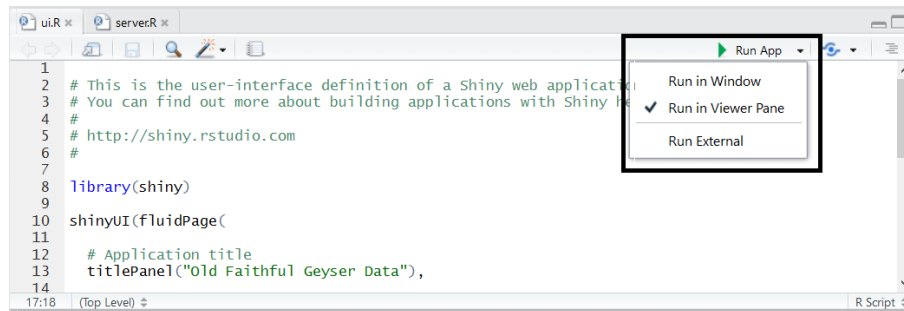
1. Plus de détails sur **shiny** à l'adresse <http://shiny.rstudio.com>.

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

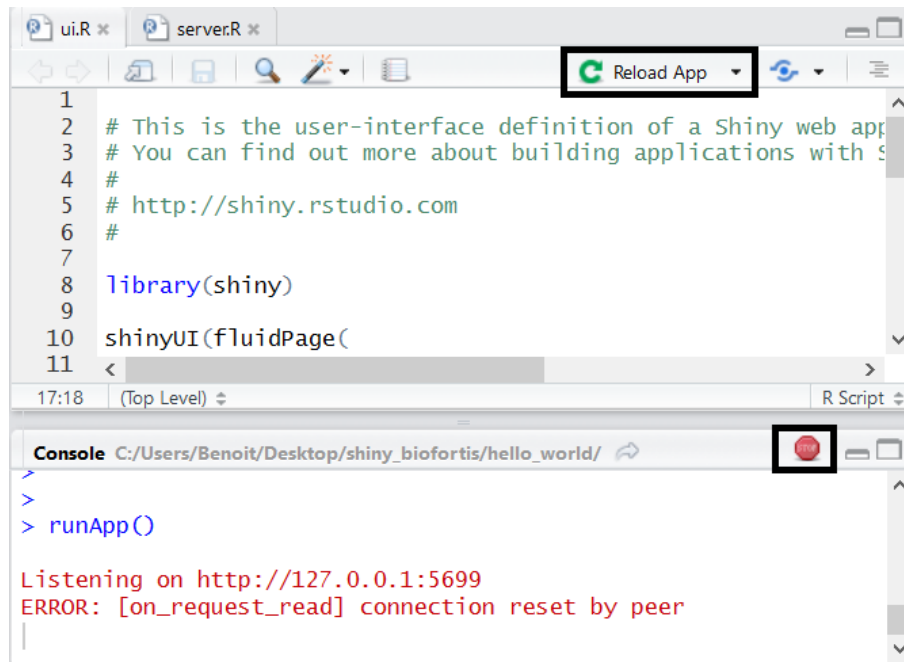
2 Première application avec shiny

Initialiser une application est simple avec **RStudio** [RStudio Team, 2015], en créant un nouveau projet : *File > New Project > New Directory > Shiny Web Application*. Cette dernière est souvent basée sur deux scripts `ui.R` et `server.R` (voir section~5, page~20), et utilise par défaut le `sidebarLayout`.

Également l'interface de cet IDE est bien pensée pour le développement d'applications avec **shiny**. On lance une application en cliquant sur le bouton **Run app**.



Une fois l'application lancée, il est possible de rafraîchir l'affichage ou d'arrêter l'application à l'aide de boutons dédiés.



Voici un exemple d'une première application, le rendu est présenté en figure~1, page~7 :

`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')
  })
})
```

Avec cette exemple simple, nous pouvons introduire et comprendre les points suivants :

Hello Shiny!

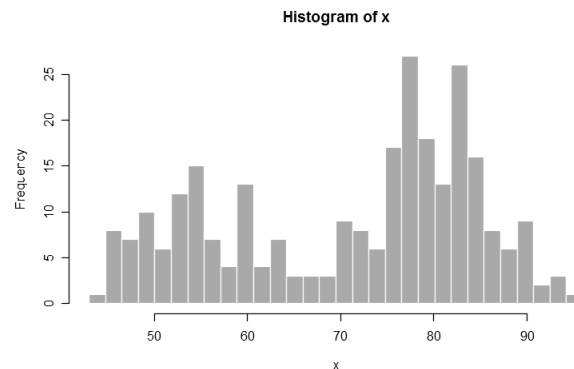
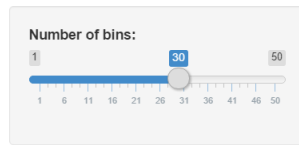


FIGURE 1 – Capture d'image d'une première application.

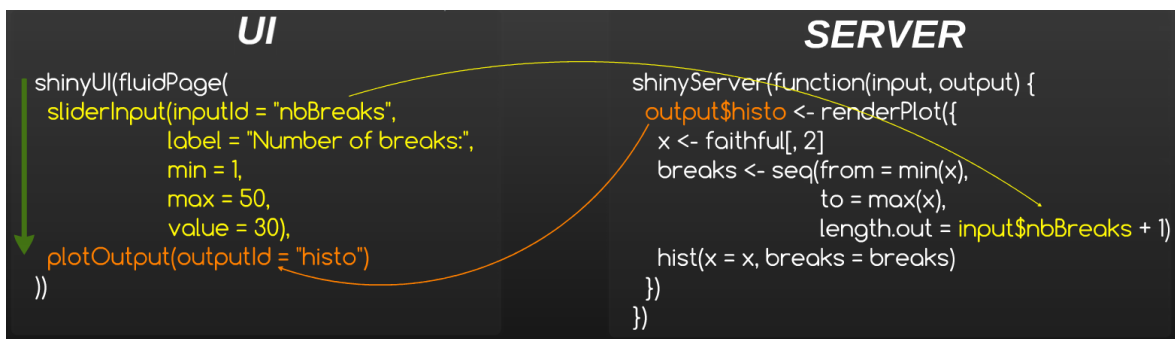


FIGURE 2 – Fonctionnement de la communication entre le serveur et l'interface.

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

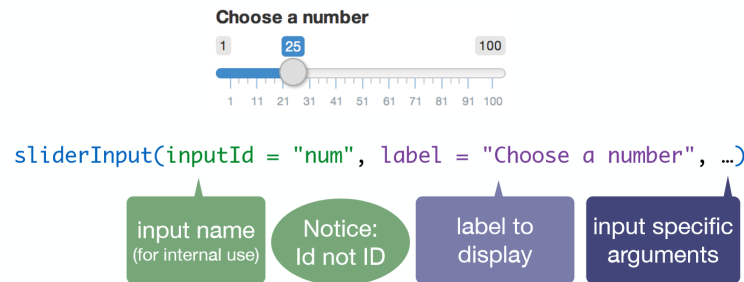
Un schéma du dialogue entre le serveur et l'interface est représenté en figure~2, page~7, il y a deux points principaux à retenir :

1. le serveur et l'interface communiquent uniquement par le biais des inputs et des outputs ;
2. par défaut, un output est mis à jour chaque fois qu'un input en lien change.

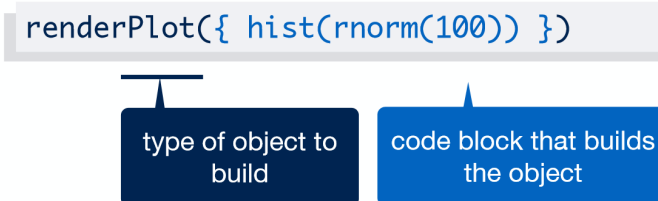
L'interface utilisateur, permet la déclaration des inputs, le placement des outputs et l'organisation visuelle de la page. Il y a deux grandes familles d'éléments :

1. `xxInput(inputId = ..., ...)` : pour définir un élément qui permet une action de l'utilisateur, ce

dernier sera accessible côté serveur via son identifiant `input$[inputID]`.



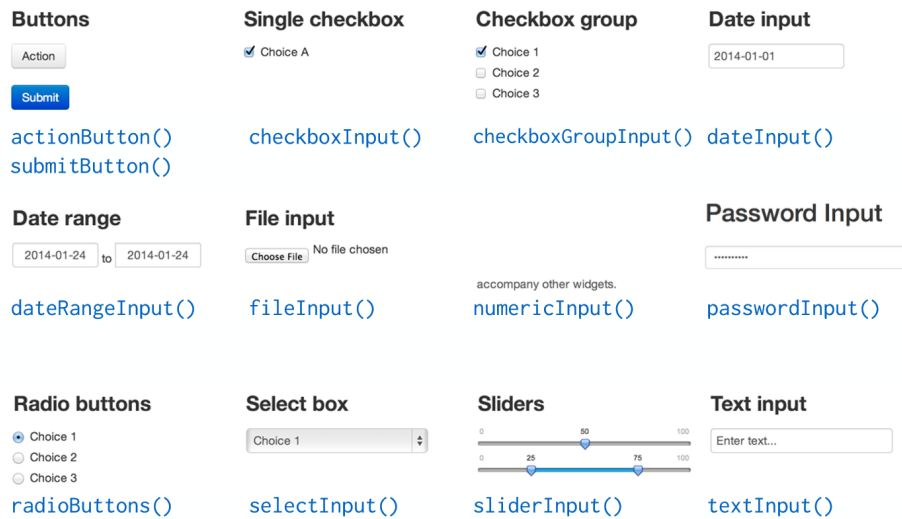
2. `xxOutput(ouputId = ...)` : pour faire référence à un output créé et défini côté serveur, il s'agit en général de graphiques et de tableaux.



Dans la partie serveur/calculs, on déclare les output, on utilise les inputs et on code en **R** ! Grâce aux expressions `renderXX({expr})`, on calcule et retourne une sortie, dépendante d'input(s), via une expression R.

3 Les inputs

Les principaux éléments d'interface sont représentés ci-après. Dans la suite, on détaille successivement le fonctionnement de certains d'entre-eux.



3.1 Valeurs numériques

3.1.1 Valeur au choix

— 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)
```

Please select a number

Value:

Class:

3.1.2 Curseur : 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)
```



3.1.3 Curseur : intervalle

— 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)
```



3.2 Les chaînes de caractère(s)

3.2.1 Valeur libre

— 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"
```



3.2.2 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; margin-bottom: 10px;"> <p>Select among the list:</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">3 ▼</div> </div>	<p>Value: <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">[1] "3"</div></p>
<div style="border: 1px solid #ccc; padding: 5px;"> <p>Select among the list:</p> <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">Third Second</div> </div>	<p>Value: <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">[1] "3" "2"</div></p> <p>Class: <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">character</div></p>

3.3 Les cases à cocher

3.3.1 Une seule sélection

— La fonction

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

— Exemple :

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

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

<div style="border: 1px solid #ccc; padding: 5px;"> <p>checkboxInput</p> <p><input checked="" type="checkbox"/> Check ?</p> </div>	<p>Value: <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">[1] TRUE</div></p> <p>Class: <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">logical</div></p>
--	--

3.3.2 Sélection multiple

— La fonction

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

— Exemple :

```
checkboxGroupInput(inputId = "idCheckGroup", label = "Please select", selected = 3,
  choices = c("First" = 1, "Second" = 2, "Third" = 3))
```

```
# For the server input$idCheckGroup is a "character" vector
```

Please select
☐ First
☒ Second
☒ Third

Value: [1] "2" "3"
 Class: character

3.3.3 Choix alternatifs

— La fonction

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

— Exemple :

```
radioButtons(inputId = "idRadio", label = "Select one", selected = 3,
             choices = c("First" = 1, "Second" = 2, "Third" = 3))
```

For the server input\$idRadio is a "character"

Select one
☐ First
☐ Second
☒ Third

Value: [1] "3"
 Class: character

3.4 Sélection de date(s)

3.4.1 Date unique

— 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"

<div style="border: 1px solid #ccc; padding: 5px; width: fit-content;"> <p>Please enter a date</p> <input type="text" value="07/12/2015"/> </div>	<p>Value: <div style="border: 1px solid #ccc; padding: 2px 5px;">[1] "2015-12-07"</div></p> <p>Class: <div style="border: 1px solid #ccc; padding: 2px 5px;">Date</div></p>
---	---

3.4.2 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

<div style="border: 1px solid #ccc; padding: 5px; width: fit-content;"> <p>Please Select a date range</p> <div style="display: flex; align-items: center; gap: 5px;"> <input type="text" value="2015-01-01"/> to <input type="text" value="2015-08-12"/> </div> </div>	<p>Value: <div style="border: 1px solid #ccc; padding: 2px 5px;">[1] "2015-01-01" "2015-08-12"</div></p> <p>Class: <div style="border: 1px solid #ccc; padding: 2px 5px;">Date</div></p>
---	--

3.5 Import d'un fichier

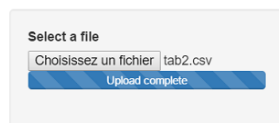
— La fonction

```
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



Value:

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

3.6 Le bouton de validation

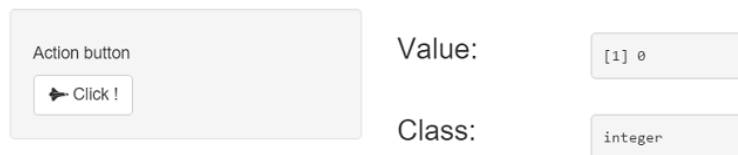
— La fonction

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

— Exemple :

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

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



Avec un peu de compétences dans les langages HTML, CSS et JavaScript, il est également possible de construire des inputs personnalisés. Un tutoriel est disponible à cette adresse : <http://shiny.rstudio.com/articles/building-inputs.html>. On peut également se référer aux deux applications suivantes à titre d'exemple :

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

4 Outputs

Il existe de nombreux outputs. Dans la suite, nous en présentons les principaux. Bien-sûr, seuls les éléments de base y figurent i.e., ceux directement disponibles en chargeant le package **shiny**, beaucoup de packages permettent d'en obtenir de nouveaux.

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

Pour assurer un bon fonctionnement, il faut assigner un identifiant (unique) à l'output pour permettre son référencement et son utilisation côté interface. On utilise une expression de type `renderXX({expr})` dont la dernière instruction doit correspondre au type d'objet retourné. Dans une telle expression, il est possible, et souvent utile, d'utiliser l'état des éléments de l'interface pour amener de la réactivité. On utilise pour cela la liste `input` et l'identifiant recherché `input[inputId]`.

```
#ui.R
selectInput("lettre", "Lettres:", LETTERS[1:3])
verbatimTextOutput(outputId = "selection")

#server.R
output$selection <- renderPrint({input$lettre})
```

4.1 Textes

4.1.1 Verbatim

Ce type de retour est utilisé pour renvoyer du texte brut (e.g., le résultat d'une sortie console).

— ui.r :

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

— server.r :

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

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


4.1.2 Texte standard

Très basique, cela permet de renvoyer du texte lorsqu'il n'est pas possible de l'indiquer directement côté interface.

— `ui.r` :

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

— `server.r` :

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

Hello shiny !

4.2 Graphiques

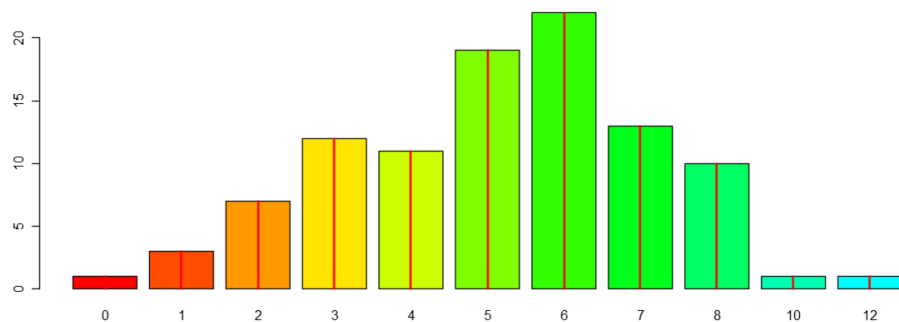
On peut renvoyer les graphiques **R** de base très simplement.

— `ui.r` :

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

— `server.r` :

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



4.3 Tableaux

Le package **shiny** donne la possibilité d'afficher les tableaux de deux façons différentes.

4.3.1 Tableau basique

Avec cette fonction, le tableau ne présentera aucune option d'affichage.

— **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

4.3.2 Tableau interactif

Avec cette fonction, il est possible d'ajouter beaucoup d'options (tri, filtre, etc.).

— **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

Previous 1 Next

4.4 Définir des éléments de l'UI côté SERVER

Dans certains cas, on souhaite définir des inputs ou des structures côté server (e.g., créer un input dépendant d'un fichier utilisateur, comme lister les colonnes présentes). Cela est possible avec les fonctions `uiOutput` et `renderUI`. Voici un 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

On peut également renvoyer un élément d'interface plus complexe, par exemple tout un `layout` ou une `fluidRow` :

— `ui.r` :

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

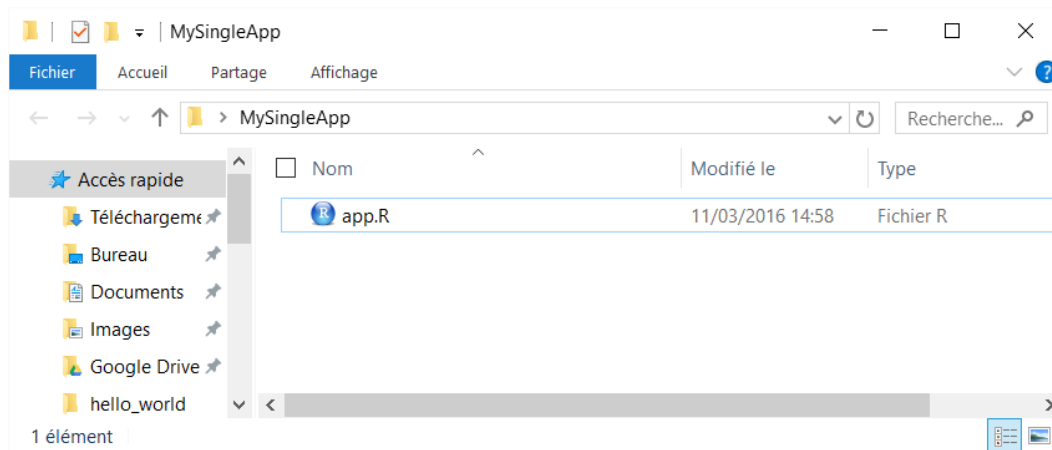
— server.r :

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

Encore une fois et c'est aussi ce qui fait la force de du package **shiny**, on peut construire des outputs avec un peu de compétences en HTML/CSS/JavaScript. Un tutoriel est disponible à cette adresse <http://shiny.rstudio.com/articles/building-outputs.html>.

5 Arborescence d'une application

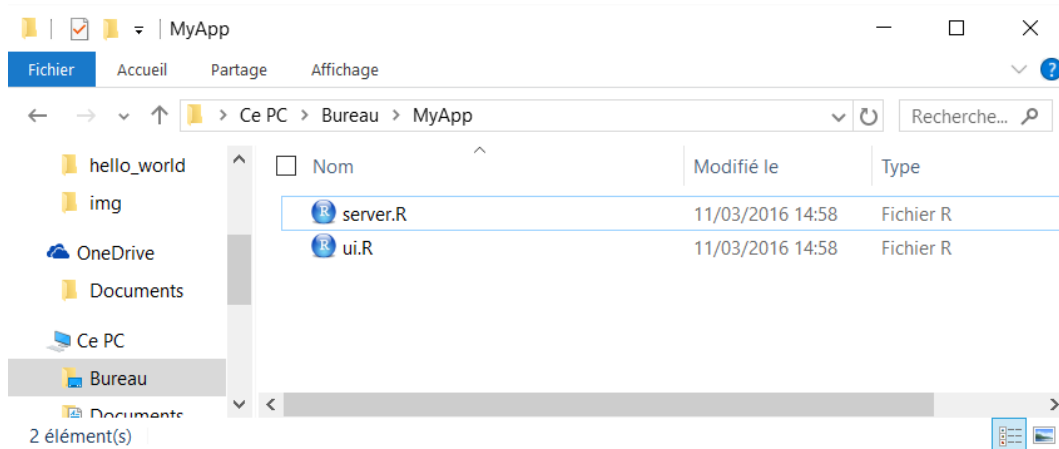
Il est possible de définir une application de différentes façons. Bien que déconseillé dans la majorité des cas, on peut n'utiliser qu'un seul script enregistré sous le nom **app.R**. Il doit alors se terminer par l'instruction **shinyApp()**. Cette utilisation doit vraiment être dédiée aux applications 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)
```

Plus généralement, on créera deux scripts. L'un dédié au serveur dans le script **server.R**, l'autre à l'interface dans le script **ui.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))})
}
```

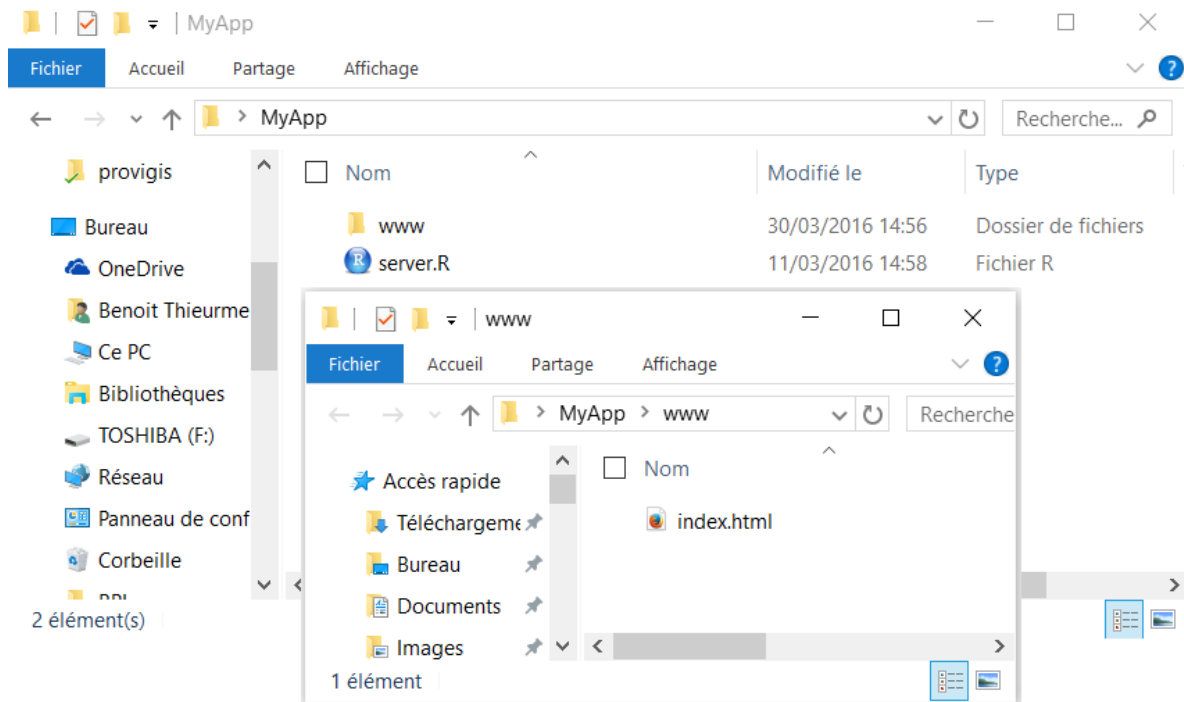


FIGURE 3 – Les éléments annexes, images, CSS, etc.

5.1 Codage de l'interface en HTML

Même si en général on code l'interface dans un script **R**, il est également possible de la coder entièrement dans un fichier **HTML**³. L'application reposera alors sur deux scripts (cf. figure~3) :

1. **server.R** dans le répertoire principal de l'application ;
2. **index.html** dans le sous-répertoire **www**.

5.2 Données/fichiers complémentaires

Le code **R** s'exécute au niveau du répertoire principal. On peut donc accéder de façon relative à tous les objets (scripts et données) présents dans le dossier de l'application. De plus, l'application côté client (comme de convention pour le web) accède à tous les éléments présents dans le dossier **www**. On illustre en figure~4 l'arborescence d'une application pour laquelle on utilise plusieurs sous-dossiers.

3. Le fonctionnement est détaillé sur l'aide officielle à l'adresse suivante <http://shiny.rstudio.com/articles/html-ui.html>.

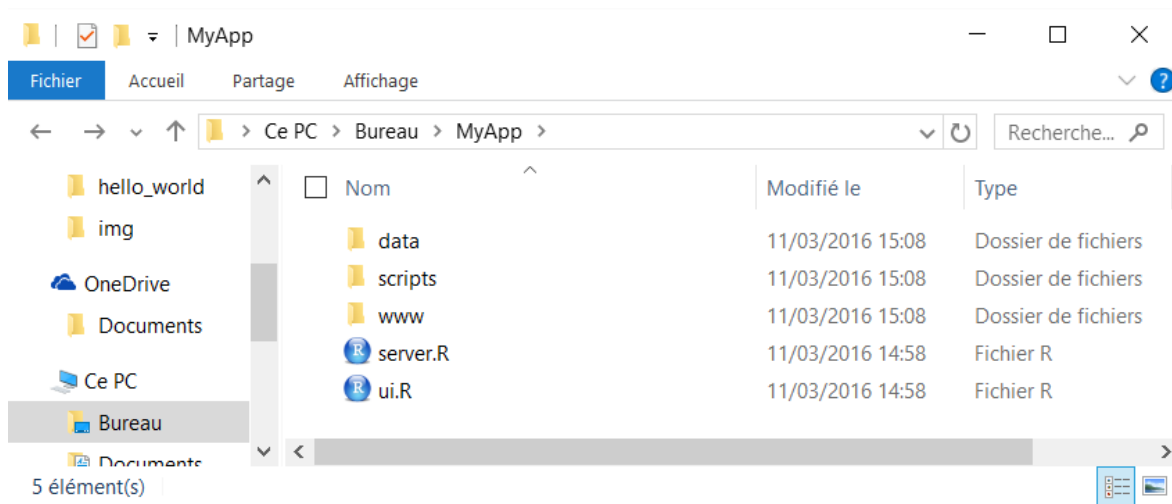


FIGURE 4 – Arborescence d'une application avec plusieurs sous-dossiers.

5.3 Partage ui <-> server

Le serveur et l'interface communiquent uniquement par le biais des inputs et des outputs. Il est possible d'ajouter un script nommé **global.R** pour partager des éléments (variables, packages, ...) entre la partie interface et la partie serveur. Tout ce qui est présent dans le script **global.R** est ainsi visible/accessible à la fois depuis **ui.R** et **server.R**. Le script **global.R** est sourcé 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 à la différence des objets de session⁴.

6 Structurer sa page

6.1 Division 1/3, 2/3

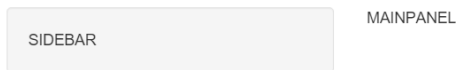
Le template basique **sidebarLayout** divise la page en deux colonnes et doit contenir (a) un **sidebarPanel** à gauche en général dédié aux inputs (b) un **mainPanel** à droite en général pour les outputs.

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

4. Plus d'informations sur la portée des variables sont accessibles sur l'aide officielle à l'adresse suivante <http://shiny.rstudio.com/articles/scoping.html>.

```
)
))
```

My first app



6.2 Le wellPanel

Comme pour le `sidebarPanel` précédent, on peut grouper 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

This screenshot shows the Shiny app interface without the use of `wellPanel`. It features a slider input labeled 'Choose a number' with a range from 1 to 100 and a current value of 25. Below the slider is a text input field labeled 'Write a title' with the value 'Histogram' entered. The elements are not grouped together in a single container.

With wellPanel

This screenshot shows the Shiny app interface with the use of `wellPanel`. The slider input 'Choose a number' and the text input 'Write a title' are now grouped together within a single, light gray rectangular container, providing a more organized and cohesive user interface.

6.3 La navigation en onglets

On peut aussi 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")
  )
)

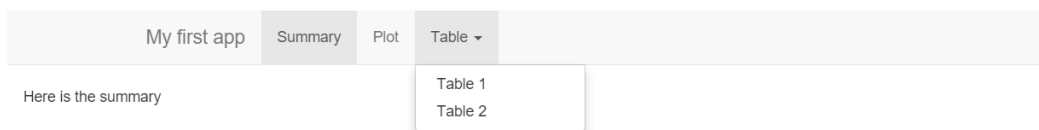
```

Nous pouvons de plus ajouter 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"))
  )
)

```

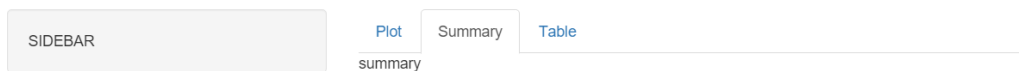


6.4 Les onglet de sélection

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

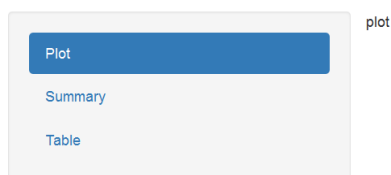
```
shinyUI(fluidPage(
  titlePanel("Old Faithful Geyser Data"), # title
  sidebarLayout(
    sidebarPanel("SIDEBAR"),
    mainPanel(
      tabsetPanel(
        tabPanel("Plot", plotOutput("plot")),
        tabPanel("Summary", verbatimTextOutput("summary")),
        tabPanel("Table", tableOutput("table"))
      )
    )
  )
))
```

My first app



La `navlistPanel` est une alternative au `tabsetPanel`, pour une disposition verticale plutôt qu'horizontale.

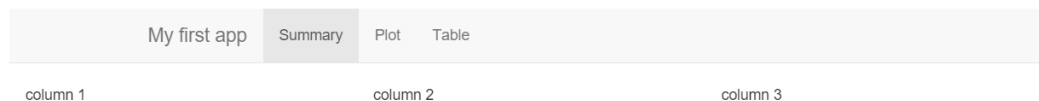
```
shinyUI(fluidPage(
  navlistPanel(
    tabPanel("Plot", plotOutput("plot")),
    tabPanel("Summary", verbatimTextOutput("summary")),
    tabPanel("Table", tableOutput("table"))
  )
))
```



6.5 Structure à la carte

Un structure très populaire dans **shiny** est la `fluidRow()`. Elle permet de diviser la page en unités verticales via les `column()`. Il est important de retenir que chaque ligne peut être divisée en **12 colonnes**. Il s'agit d'un dimensionnement relatif à la taille de l'écran côté client. Le rendu final de la page est automatiquement adapté 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"),
  ))
```



6.6 Inclure du 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"      "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"

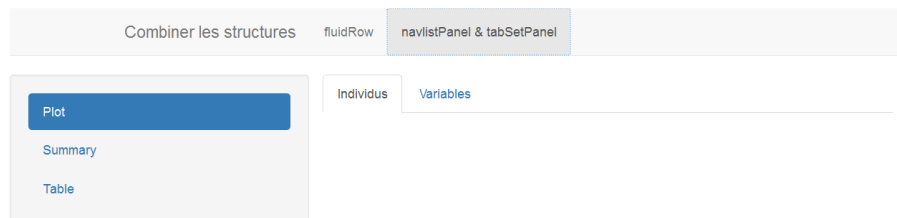
```
tags$a(href = "www.rstudio.com", "RStudio")
```



Il est également possible de faire appel à du code **HTML** directement en utilisant la fonction du même nom :

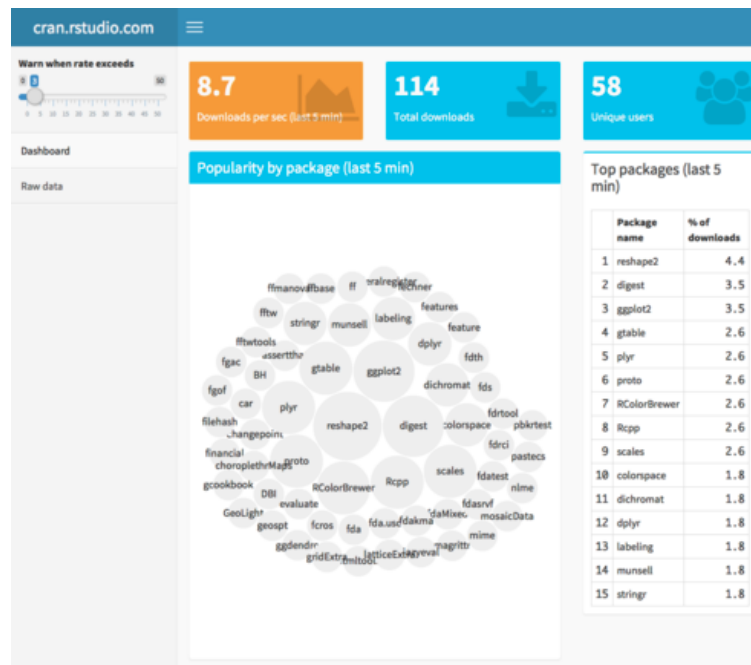
```
fluidPage(
  HTML("<h1>My Shiny App</h1>")
)
```

Le package **shiny** permet une personnalisation infinie car toutes les structures peuvent s'utiliser en même temps !



6.7 Un package, une interface : shinydashboard

Le package **shinydashboard** [Chang and Borges Ribeiro, 2018]⁵ propose d'autres fonctions pour créer des tableaux de bords :



7 Aspect général, personnalisation et réactivité

7.1 Customisation avec du CSS

Shiny utilise le thème Bootstrap (cf. <http://getbootstrap.com>) pour la partie **CSS**. Comme pour un développement web « classique », nous pouvons modifier les propriétés **CSS** de trois façons :

- en faisant un lien vers un fichier **.css** externe et en ajoutant des feuilles de style dans le répertoire **www**⁶ ;
- en ajoutant des propriétés **CSS** en en-tête i.e., dans le header **HTML** ;
- en écrivant individuellement des propriétés **CSS** dans les éléments.

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

5. Cf. page de documentation <https://rstudio.github.io/shinydashboard>

6. Quelques exemples sont à disposition grâce au package **shinythemes** [Chang, 2016] (cf. <http://rstudio.github.io/shinythemes>).

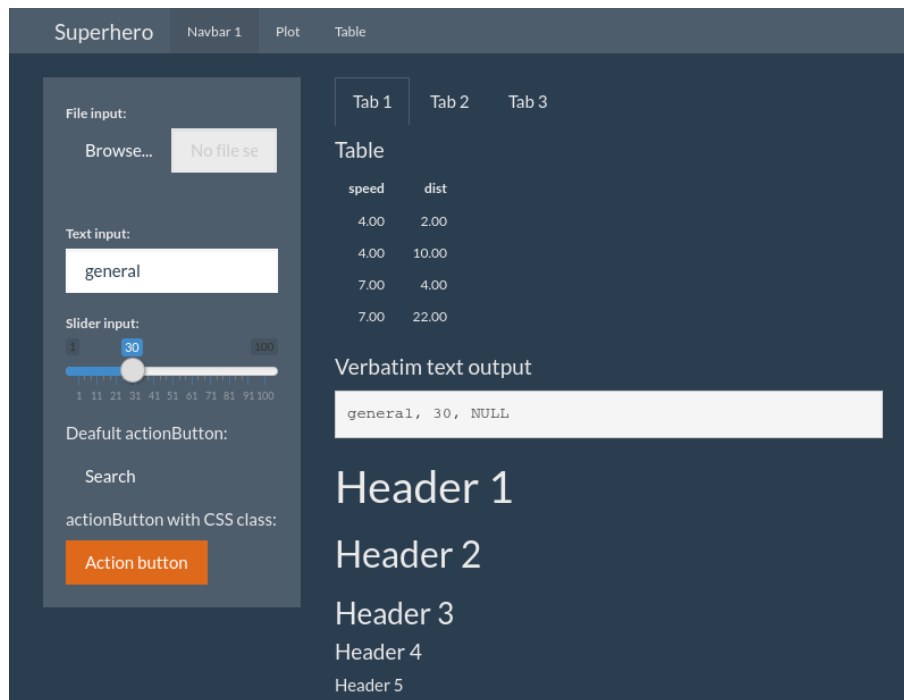


FIGURE 5 – Exemple du thème `superhero`. Beaucoup d'autres sont disponibles à l'adresse <http://bootswatch.com>.

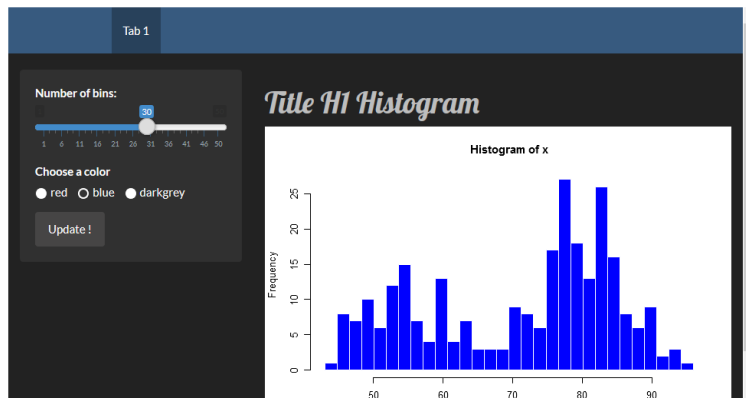
On peut aussi utiliser complètement un fichier `.css` externe (e.g., voir figure~5). Dans ce cas, il y a deux façons d'y faire appel :

- utiliser l'argument `theme` dans `fluidPage`;
- utiliser un tag 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
)
```

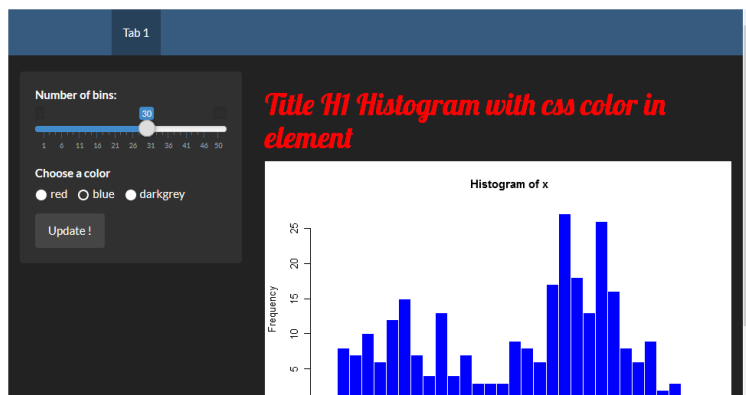
Si l'on souhaite ajouter une propriété en en-tête, elle sera prioritaire sur le **CSS** externe. On utilise alors les tags HTML : `tags$head` et `tags$style`

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



On peut également passer directement du CSS aux éléments HTML :

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



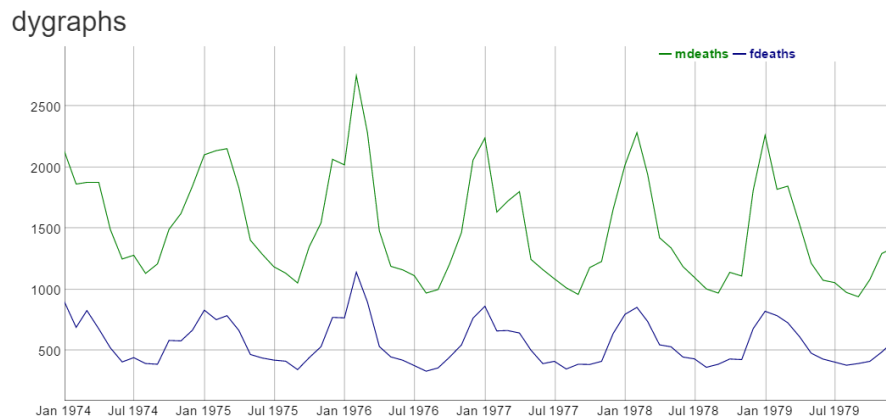
7.2 Graphiques interactifs

Avec notamment l'arrivée du package **htmlwidgets** [Vaidyanathan et al., 2018]⁷, de plus en plus de fonctionnalités JavaScript sont accessibles sous **R**. Voici quelques librairies pouvant s'avérer :

- **dygraphs** (time series) [Vanderkam et al., 2018] (cf. <http://rstudio.github.io/dygraphs>)
- **DT** (interactive tables) [Xie, 2018] (cf. <http://rstudio.github.io/DT>)
- **Leaflet** (maps) [Cheng et al., 2018] (cf. <http://rstudio.github.io/leaflet>)
- **d3heatmap** [Cheng and Galili, 2018] (cf. <https://github.com/rstudio/d3heatmap>)
- **threejs** [Lewis, 2017] (3d scatter & globe) (cf. <http://bwlewis.github.io/rthreejs>)
- **rAmCharts** [Thieurmél et al., 2018] (cf. http://datastorm-open.github.io/introduction_ramcharts)
- **visNetwork** [?] (cf. <http://datastorm-open.github.io/visNetwork>)
- ...

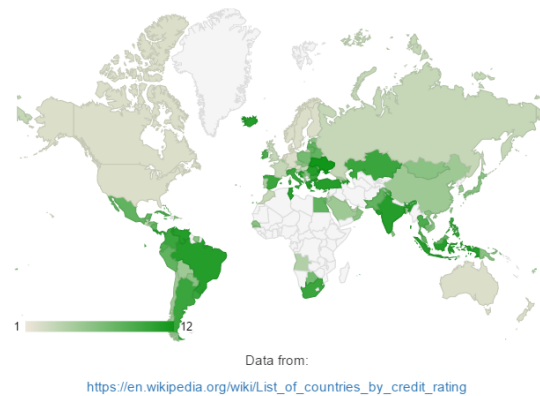
Tous ces packages sont utilisables simplement dans **shiny**. Ils contiennent les deux fonctions nécessaires **renderXX** et **xxOutput**. Par exemple avec le package **dygraphs** [Vanderkam et al., 2018] :

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



7. Pour de nombreux exemples, on peut jeter un oeil sur la galerie à l'adresse <http://gallery.htmlwidgets.org>. Également, on peut se référer à l'aide en ligne <https://bookdown.org/yihui/rmarkdown>.

googleVis Example



8 Pour une meilleure maîtrise de la réactivité

8.1 Isolation

Par défaut, les outputs et les expressions réactives se mettent à jour **automatiquement** quand un des inputs présents dans le code change de valeur. Cependant et dans certains cas, on aimerait pouvoir contrôler un peu cela. Par exemple, en utilisant un bouton (e.g., `actionButton`) pour déclencher le calcul des sorties. Un input peut être isolé ainsi `isolate(input$[id])`. On peut aussi isoler une expression avec la notation suivante `isolate({expr})`.

Prenons l'exemple d'une interface simple avec trois inputs : `color` et `bins` pour l'histogramme, et un `actionButton` :

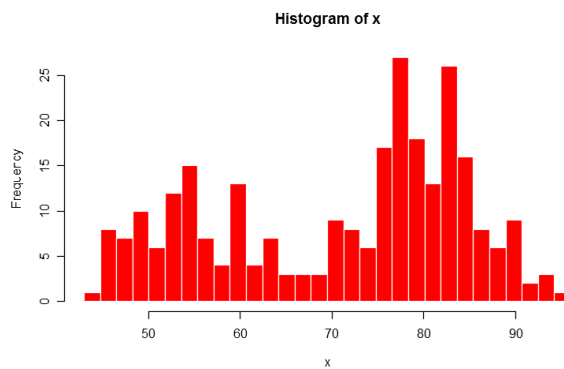
```
shinyUI(fluidPage(
  titlePanel("Isolation"),
  sidebarLayout(
    sidebarPanel(
      radioButtons(inputId = "col", label = "Choose a color", inline = TRUE,
        choices = c("red", "blue", "darkgrey")),
      sliderInput("bins", "Number of bins:", min = 1, max = 50, value = 30),
      actionButton("go_graph",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

8.1.1 Isolation par expression

On isole tout le code sauf l'**actionButton**. L'histogramme sera mis à jour **uniquement** quand l'utilisateur cliquera sur le bouton :

```
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')
    })
  })
})
```

Isolation



On isole tout le code sauf l'**actionButton** et l'**input\$color**. L'histogramme sera mis à jour **soit** quand l'utilisateur cliquera sur le bouton **soit** quand il changera la couleur :

— **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')
  })
})
```

```
})
```

8.1.2 Isolation par input

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

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

8.2 Les 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))
})
```

Une expression réactive va nous faire gagner du temps et de la mémoire, *RAM*. Cependant, on conseille d'utiliser des expressions réactives seulement quand cela dépend d'inputs⁸. Une expression réactive se comporte **comme un output**. Sa valeur est mise à jour chaque fois qu'un input présent dans le code change. Également, elle se comporte **comme un input** dans un `renderXX` dans la mesure où l'output est mis à jour quand l'expression réactive change. On notera qu'on récupère la valeur d'une expression réactive de la même manière que l'on ferait **appel à une fonction sans argument** i.e., avec des parenthèses.

Il existe une alternative à l'utilisation de `reactive` avec `reactiveValues`. Cela permet d'initialiser une liste d'objets réactifs, un peu comme la liste des inputs dont on va pouvoir par la suite modifier la valeur des objets avec des `observe` ou des `observeEvent`.

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

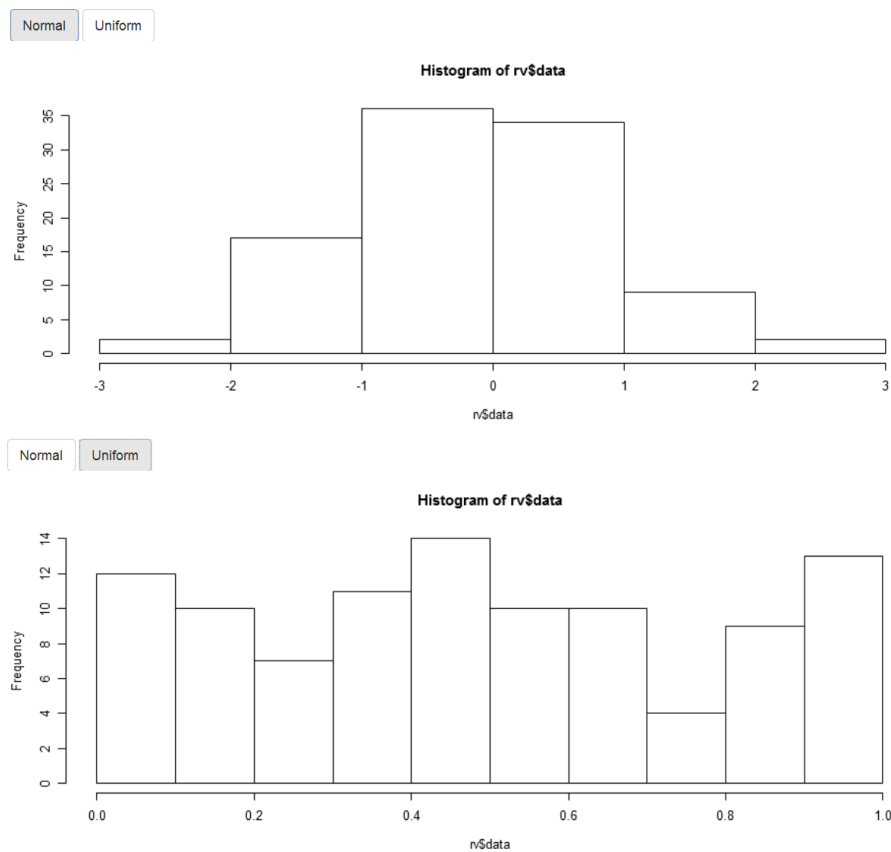
shinyApp(ui = fluidPage(
  actionButton(inputId = "norm", label = "Normal"),
```

8. Plus d'informations sur la portée des variables sont accessibles sur l'aide officielle à l'adresse suivante <http://shiny.rstudio.com/articles/scoping.html>.

```

  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) })
})

```



8.3 Les fonctions de mise à jour

Il existe une série de fonctions pour mettre à jour les inputs et certaines structures. Elles commencent par `update...` et 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)
    )
  })
})

```

Observer

Choose a dataset: ☒ cars ☐ iris ☐ quakes

Choose a column:

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:

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

Exemple sur des onglets :

Il faut rajouter un id dans la structure

```

shinyUI(
  navbarPage(

```



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

)
)

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

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")
})
```

8.4 Les élément d'interface conditionnels

Il est possible d'afficher conditionnellement certains éléments avec l'appel suivant `conditionalPanel(condition = [...],)`. La condition peut dépendre des inputs ou des outputs. Elle doit être rédigée en javascript :

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

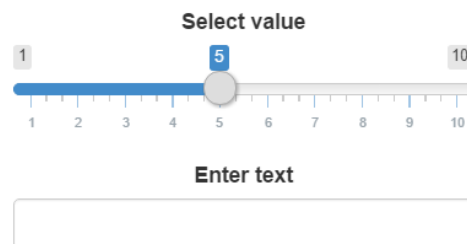
library(shiny)
shinyApp(
  ui = fluidPage(
    fluidRow(
      column(
        width = 4,
        align = "center",
        checkboxInput("checkbox", "View other inputs", value = FALSE)
      ),
      column(
        width = 8,
        align = "center",
        conditionalPanel(
          condition = "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

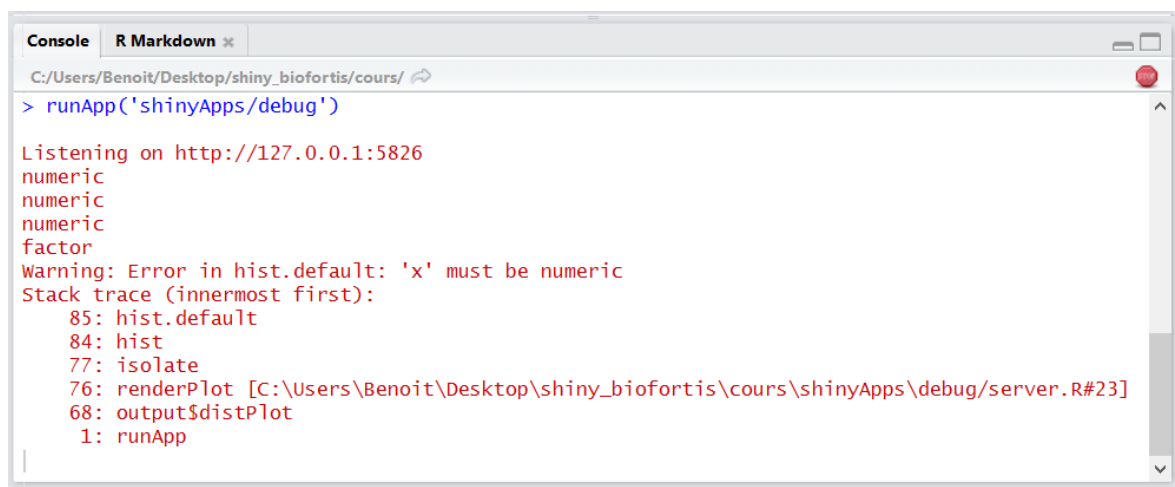


9 Débogage

9.1 Affichage console

Un des premiers niveaux de débogage est l'utilisation de *print console* au sein de l'application shiny. Cela permet d'afficher des informations lors du développement et/ou de l'exécution de l'application. C'est un peu déconseillé car il est parfois difficile et souvent laborieux de les supprimer à la fin du développement. 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)
})
```

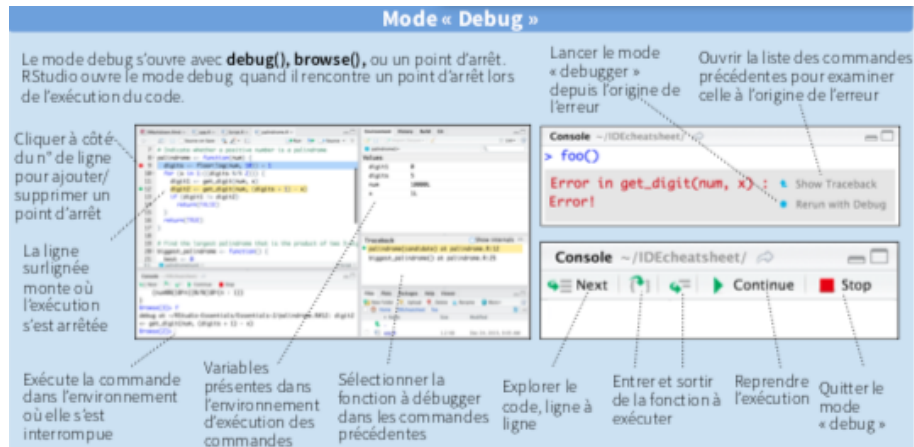


On peut aussi intégrer le lancement d'un `browser()` à n'importe quel moment pour observer les différents objets et avancer pas-à-pas

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

Cependant, il ne faut pas oublier de l'enlever une fois le développement terminé. Au cours du développement, la meilleure solution est souvent le recours aux points d'arrêt. Ils permettent de suivre facilement le déroulement de l'exécution et échappent par nature aux problème de suppression

en fin de projet car il ne n'affectent pas la console.



9.2 Lancement automatique d'un browser

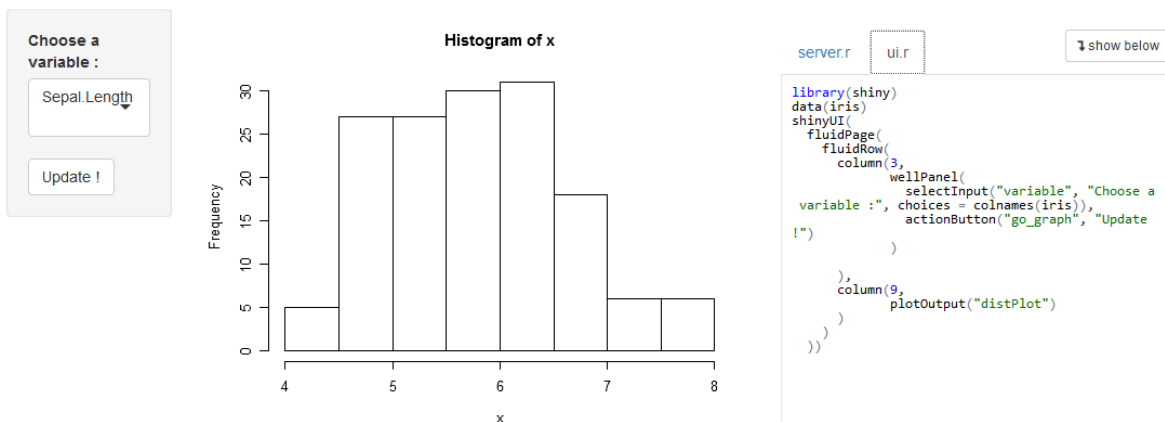
L'option `options(shiny.error = browser)` permet de lancer un `browser()` automatiquement lors de l'apparition d'une erreur

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

9.3 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")
```



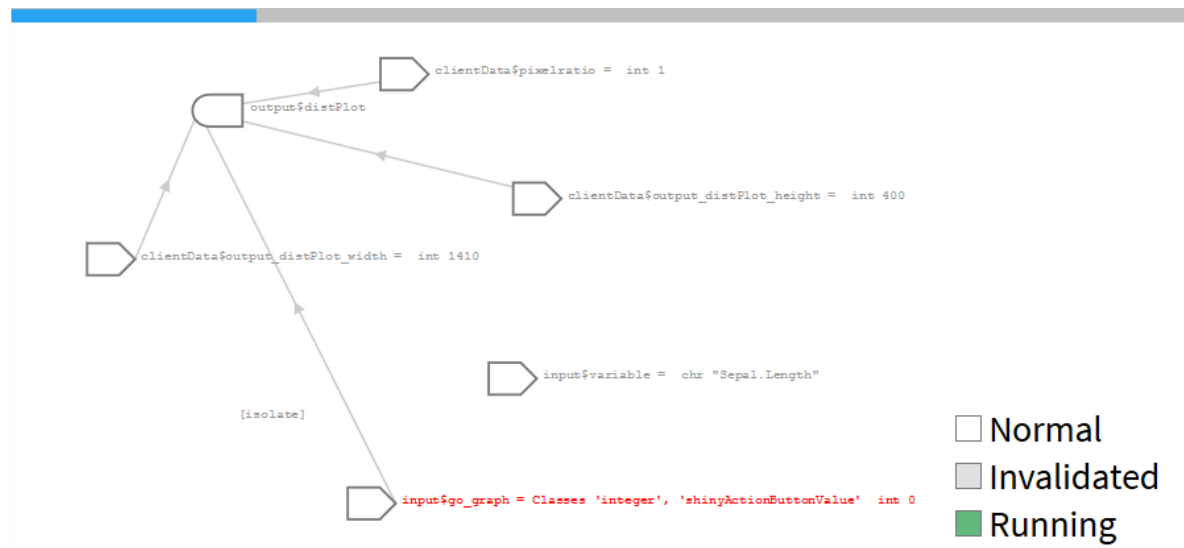
9.4 Reactive log

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)
})
```



9.5 Communication client/server

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

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

```

> 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"}}

```

9.6 Traçage des erreurs

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

`options(shiny.fullstacktrace = TRUE)`

```

> 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]
R#35]
 82: contextFunc
 81: env$runWith
 80: withReactiveDomain
 79: ctx$run
 78: ...

```

10 Conclusion

10.1 Quelques bonnes pratiques


- Préférer l'underscore (_) au point (.) comme séparateur dans le nom des variables. En effet, le . peut amener de mauvaises interprétations 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** [Ushey et al., 2018] (cf. <https://rstudio.github.io/packrat>)
- Mettre toute la **partie « calcul »** dans des **fonctions/un package** et effectuer des tests avec **testthat** [Wickham, 2011] (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)
})
```

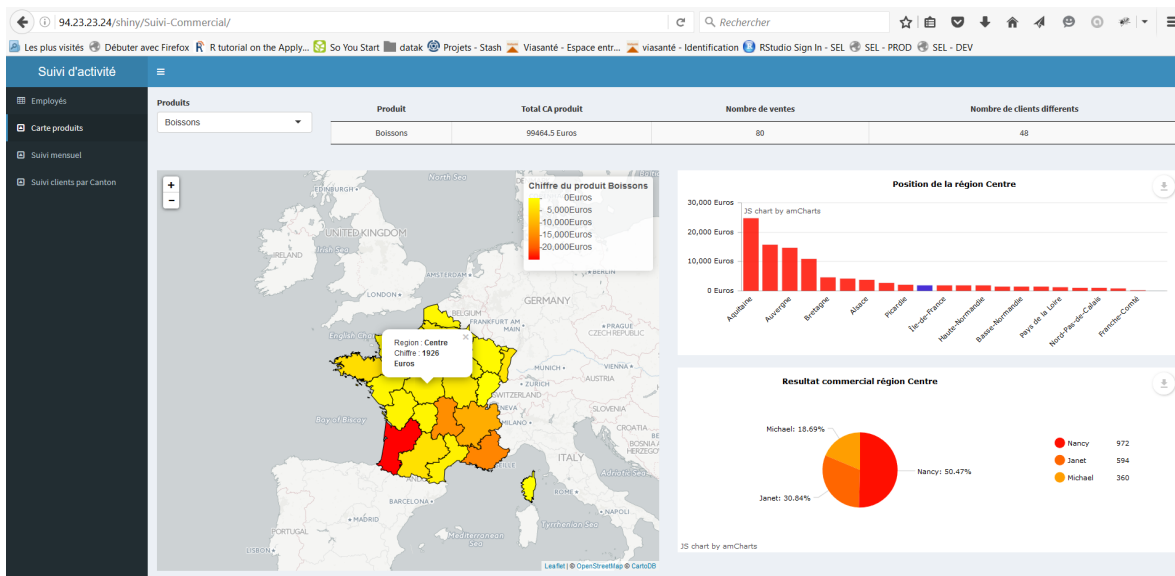
10.2 Quelques mots sur shiny-server

On peut déployer en interne nos applications shiny en installant un shiny-server (cf. <https://www.rstudio.com/products/shiny/shiny-server2>). C'est uniquement disponible sur linux : ubuntu 12.04+, RedHat/CentOS 5+, SUSE Enterprise Linux 11+. La version gratuite permet de déployer plusieurs applications **shiny**. La version payante propose entre autres l'authentification des utilisateurs, une optimisation des ressources par applications (nombre de coeurs, mémoire, etc.), un monitoring. Une fois le serveur installé, il suffit de déposer les applications dans le répertoire dédié, et elles deviennent directement accessibles via l'adresse **server:port_ou_redirection/nom_du_dossier**.

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

Index of /apps/

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



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

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

10.3 Références / Tutoriels / Exemples

- <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>

NB : Ce document a été rédigé avec **rmarkdown** [Allaire et al., 2018].

Références

JJ Allaire, Yihui Xie, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, Hadley Wickham, Joe Cheng, and Winston Chang. *rmarkdown : Dynamic Documents for R*, 2018. URL <https://CRAN.R-project.org/package=rmarkdown>. R package version 1.10.

Winston Chang. *shinythemes : Themes for Shiny*, 2016. URL <https://CRAN.R-project.org/package=shinythemes>. R package version 1.1.1.

Winston Chang and Barbara Borges Ribeiro. *shinydashboard : Create Dashboards with 'Shiny'*, 2018. URL <https://CRAN.R-project.org/package=shinydashboard>. R package version 0.7.0.

Winston Chang, Joe Cheng, JJ Allaire, Yihui Xie, and Jonathan McPherson. *shiny : Web Application Framework for R*, 2018. URL <https://CRAN.R-project.org/package=shiny>. R package version 1.1.0.

- Joe Cheng and Tal Galili. *d3heatmap : Interactive Heat Maps Using 'htmlwidgets' and 'D3.js'*, 2018. URL <https://CRAN.R-project.org/package=d3heatmap>. R package version 0.6.1.2.
- Joe Cheng, Bhaskar Karmabelkar, and Yihui Xie. *leaflet : Create Interactive Web Maps with the JavaScript 'Leaflet' Library*, 2018. URL <https://CRAN.R-project.org/package=leaflet>. R package version 2.0.2.
- B. W. Lewis. *threejs : Interactive 3D Scatter Plots, Networks and Globes*, 2017. URL <https://CRAN.R-project.org/package=threejs>. R package version 0.3.1.
- R Core Team. *R : A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018. URL <https://www.R-project.org/>.
- RStudio Team. *RStudio : Integrated Development Environment for R*. RStudio, Inc., Boston, MA, 2015. URL <http://www.rstudio.com/>.
- Benoit Thieurmél, Antanas Marcelionis, Jeffery Petit, Elena Salette, and Titouan Robert. *rAmCharts : JavaScript Charts Tool*, 2018. URL <https://CRAN.R-project.org/package=rAmCharts>. R package version 2.1.8.
- Kevin Ushey, Jonathan McPherson, Joe Cheng, Aron Atkins, and JJ Allaire. *packrat : A Dependency Management System for Projects and their R Package Dependencies*, 2018. URL <https://CRAN.R-project.org/package=packrat>. R package version 0.4.9-3.
- Ramnath Vaidyanathan, Yihui Xie, JJ Allaire, Joe Cheng, and Kenton Russell. *htmlwidgets : HTML Widgets for R*, 2018. URL <https://CRAN.R-project.org/package=htmlwidgets>. R package version 1.3.
- Dan Vanderkam, JJ Allaire, Jonathan Owen, Daniel Gromer, and Benoit Thieurmél. *dygraphs : Interface to 'Dygraphs' Interactive Time Series Charting Library*, 2018. URL <https://CRAN.R-project.org/package=dygraphs>. R package version 1.1.1.6.
- Hadley Wickham. *testthat : Get started with testing*. *The R Journal*, 3 :5–10, 2011. URL https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf.
- Yihui Xie. *DT : A Wrapper of the JavaScript Library 'DataTables'*, 2018. URL <https://CRAN.R-project.org/package=DT>. R package version 0.4.