

Interfaces graphiques avec R et shiny

1 - Les concepts de base de shiny

Robin Cura & Lise Vaudor
d'après L. Vaudor : [Formation shiny_\(2018\)](#).

16/10/2018

École Thématique GeoViz 2018

Sommaire

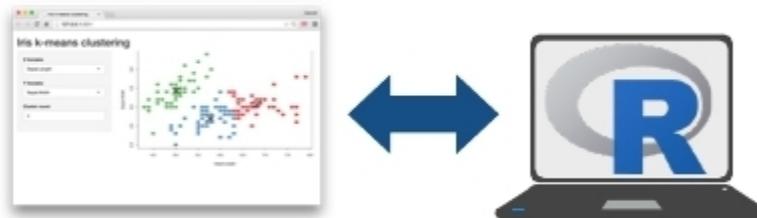
- Organisation générale d'une application shiny
- Les composants/widgets
 - Les Inputs
 - Les Outputs
- Mise en page
 - Layouts
 - Panels
- Réactivité : résumé
 - Base
 - Sortie réactive
 - Fonctions réactives
 - Observation d'éléments réactifs
 - Bloquer la réactivité
- Applications
 - Démonstration
 - Exercice

Shiny: qu'est-ce que c'est?

Shiny, c'est un **package R** qui facilite la construction d'**applications web** interactives depuis R.

Basics

A **Shiny** app is a web page (**UI**) connected to a computer running a live R session (**Server**)



Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

CC BY RStudio

Les utilisateurs peuvent simplement manipuler une application "clique-boutons" pour exécuter et afficher des résultats fournis par du code R.

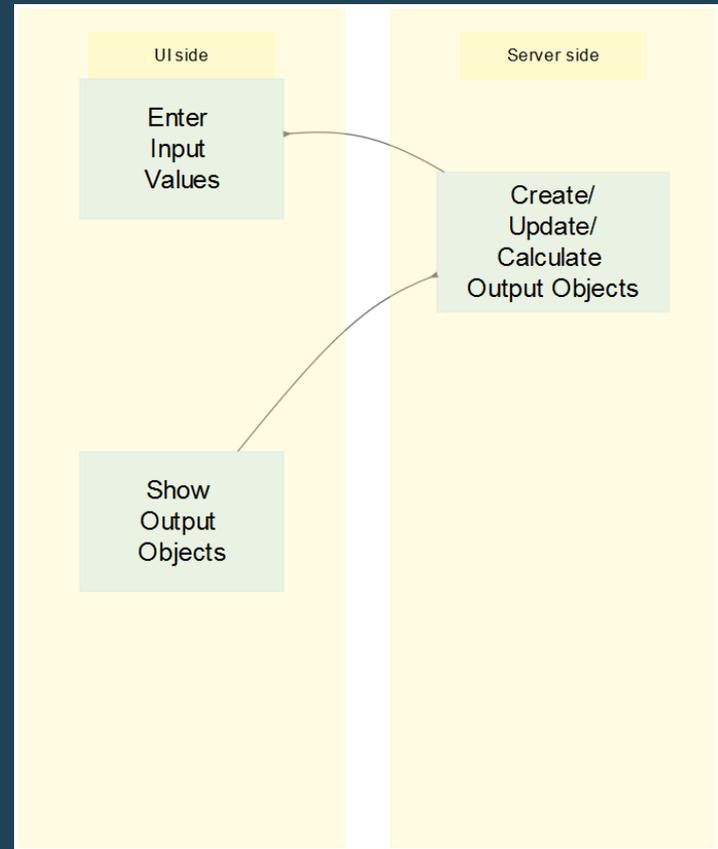
Shiny: qu'est-ce que c'est?

- Les résultats fournis sont **réactifs**, c'est-à-dire que quand l'utilisateur fournit une nouvelle valeur d'**input** (= entrée) via un **widget** (= window gadget), les codes R qui dépendent de cet input sont réexécutés et leurs sorties (**output**) affichées.
 - Voir les exemples dans la galerie shiny :
<http://shiny.rstudio.com/gallery/>
- Remarquez que toutes ces applications s'affichent dans un navigateur internet : elles sont donc rendues en **html**.
- Shiny permet en fait de produire des applications en HTML sans faire appel à ce langage, simplement avec du code R.
- Une connaissance de **html**, **css** et **javascript** reste un plus pour produire des applications plus personnalisées.

Shiny: qu'est-ce que c'est?

Une Shiny App se structure en deux parties:

- un côté **UI** qui regroupe tous les éléments de **mise en forme et d'affichage** de l'interface utilisateur elle-même (affichage des **inputs** et des **outputs**)
- un côté **Server** où sont exécutés les codes R qui servent à **produire les outputs** (graphiques, tables, traitements, etc.) et à les **mettre à jour** en cas de changement dans les valeurs d'**inputs**



Introduction: démarrer avec le template

Pour construire votre première appli Shiny, vous pouvez vous aider du **modèle** (*template*) fourni par RStudio, en faisant **File -> New file -> Shiny Web App -> Multiple File**.

Deux fichiers sont alors créés: **server.R**, et **ui.R**.

- La partie **serveur** contient l'ensemble du code R qui doit être exécuté par l'appli pour fournir les sorties.
- La partie **ui** (= user interface) contient les instructions de construction/mise en forme de l'interface utilisateur.

Vous pouvez ouvrir l'un ou l'autre, et cliquer sur le bouton **Run App** en haut à droite de la partie "script" de l'interface RStudio pour **lancer l'application**.

APP TEMPLATE

Begin writing a new app with this template. Preview the app by running the code at the R command line.



```
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines **ui** and **server** into an app. Wrap with **runApp()** if calling from a sourced script or inside a function.

Construire une app (1)

Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

```
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

```
# server.R
function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
```

ui.R contains everything you would save to ui.

server.R ends with the function you would save to server.

No need to call **shinyApp()**

-  UI:

- Ajoutez des **éléments d'entrée** à l'interface avec les fonctions de type ***Input ()**
- Ajoutez des **éléments de sortie** à l'interface avec les fonctions de type ***Output ()**

-  server:

- Encapsulez le code utilisé pour créer l'output dans une fonction de type **render* ()**
- **Assignez à l'output** la sortie de **render* ()**.

Construire une app (2)

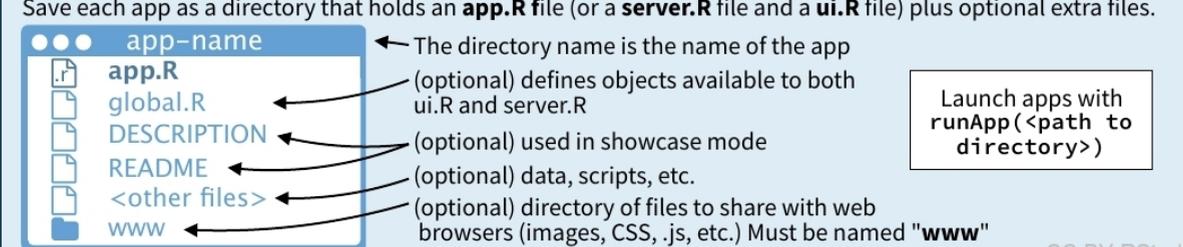
- Deux structures de base sont possibles pour les apps: soit avoir tout réuni dans un même script (**app.R**), soit séparer la partie **ui** et la partie **server** dans deux fichiers (**ui.R** et **server.R**). C'est cette deuxième solution que nous allons privilégier ici.

<pre># ui.R fluidPage(numericInput(inputId = "n", "Sample size", value = 25), plotOutput(outputId = "hist"))</pre>	<p>ui.R contains everything you would save to ui.</p> <p>server.R ends with the function you would save to server.</p> <p>No need to call shinyApp()</p>
<pre># server.R function(input, output) { output\$hist <- renderPlot({ hist(rnorm(input\$n)) }) }</pre>	

- Dans cet exemple, **ui** est un *objet de type UI* issu de l'appel à une *fonction*, ici **fluidPage()**. Les différents éléments passés à **fluidPage()** sont donc des *arguments*: ils sont séparés par des virgules.
- Dans **server** on définit une *fonction*, avec **input** et **output** comme arguments. Le corps de cette fonction s'écrit donc comme une suite de lignes de commandes: les commandes sont séparées par des retours à la ligne.

Construire une app (3)

Save each app as a directory that holds an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.



The diagram shows a directory named 'app-name' containing the following files and folders:

- app.R**: The main application file.
- global.R**: (optional) defines objects available to both ui.R and server.R.
- DESCRIPTION**: (optional) used in showcase mode.
- README**: (optional) data, scripts, etc.
- <other files>**: (optional) directory of files to share with web browsers (images, CSS, .js, etc.). Must be named "www".
- www**: (optional) directory of files to share with web browsers (images, CSS, .js, etc.). Must be named "www".

Launch apps with `runApp(<path to directory>)`

CC BY RStudio

- Le répertoire qui contient votre application doit être construit d'une manière qui facilite son déploiement sur un serveur distant :
 - **ui.R**, **server.R**, et éventuellement **global.R** à la racine
 - des sous-dossiers pour (par exemple) les données, des scripts, etc.
 - un dossier **www** qui permet de faire référence à des éléments utiles au navigateur web, *qui ne sont pas issus de calculs de R* (images, photos, logos, feuilles de style css, etc.)
- Les commandes du fichier **global.R** sont exécutées dans un environnement "**global**". C'est-à-dire que les packages qu'on y charge ou les objets qu'on y crée seront disponibles **à la fois pour les parties UI et Server**. Les commandes de **global** sont donc exécutées une fois pour toutes (c'est-à-dire, une fois par session, et avant toute autre chose) au lancement de l'application.

Partie 2 : Inputs et outputs

Inputs

- Les **inputs** sont les composants (*widgets*) de l'interface graphique qui permettent aux utilisateurs de fournir des valeurs aux paramètres d'entrée.

Action button

Action

Current Value:

```
[1] 0  
attr(,"class")  
[1] "integer"      "shinyActionButtonValue"
```

See Code

Single checkbox

Choice A

Current Value:

```
[1] TRUE
```

See Code

Checkbox group

Choice 1
 Choice 2
 Choice 3

Current Values:

```
[1] "1"
```

See Code

Date input

2014-01-01

Current Value:

```
[1] "2014-01-01"
```

See Code

Date range

2018-10-09 to 2018-10-09

Current Values:

```
[1] "2018-10-09" "2018-10-09"
```

See Code

File input

Browse... No file selected

Current Value:

```
NULL
```

See Code

Inputs

- Les **inputs** sont les composants (*widgets*) de l'interface graphique qui permettent aux utilisateurs de fournir des valeurs aux paramètres d'entrée.

The image displays six examples of Shiny input widgets arranged in a 2x3 grid. Each example includes a visual representation of the widget and a text box showing the R code used to create it.

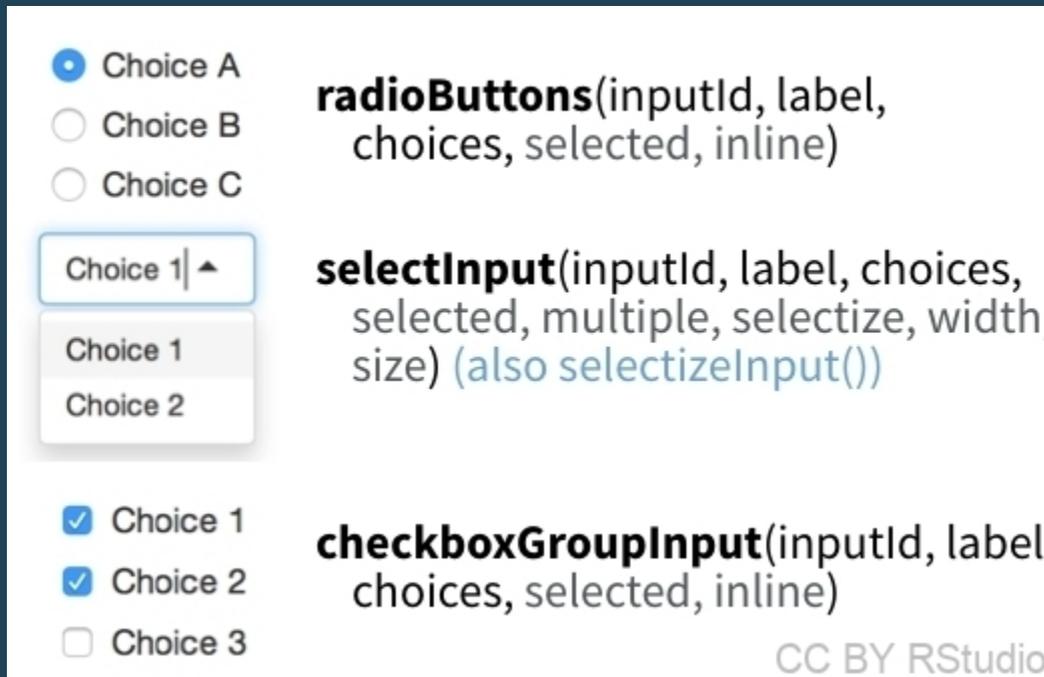
- Numeric input:** A text box containing the number '1'. Below it, a text box shows the R code: `[1] 1`.
- Radio buttons:** Three radio buttons labeled 'Choice 1', 'Choice 2', and 'Choice 3', with 'Choice 1' selected. Below it, a text box shows the R code: `[1] "1"`.
- Select box:** A dropdown menu with 'Choice 1' selected. Below it, a text box shows the R code: `[1] "1"`.
- Slider:** A horizontal slider with a range from 0 to 100 and a current value of 50. Below it, a text box shows the R code: `[1] 50`.
- Slider range:** A horizontal slider with a range from 0 to 100 and two current values, 25 and 75. Below it, a text box shows the R code: `[1] 25 75`.
- Text input:** A text box with the placeholder text 'Enter text...'. Below it, a text box shows the R code: `[1] "Enter text..."`.

- Vous pouvez avoir un aperçu de l'ensemble des inputs disponibles pour Shiny dans la galerie des *widgets* :
<https://shiny.rstudio.com/gallery/widget-gallery.html>

Inputs - Choix multiple

Pour choisir **une ou plusieurs valeurs** parmi plusieurs valeurs **prédéfinies**, plusieurs widgets sont disponibles:

- `radioButtons()` et `selectInput()` permettent de choisir **une valeur**.
- `checkboxGroupInput()` et `selectInput(..., multiple=TRUE)` permettent de choisir **plusieurs valeurs**.



The screenshot displays three different input widgets side-by-side, each with its corresponding R function signature. The first widget is a radio button group with three options: Choice A (selected), Choice B, and Choice C. The second widget is a select input dropdown menu showing 'Choice 1' and a list of options 'Choice 1' and 'Choice 2'. The third widget is a checkbox group with three options: Choice 1 (checked), Choice 2 (checked), and Choice 3 (unchecked). The text 'CC BY RStudio' is visible in the bottom right corner of the screenshot.

radioButtons(inputId, label, choices, selected, inline)

selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also `selectizeInput()`)

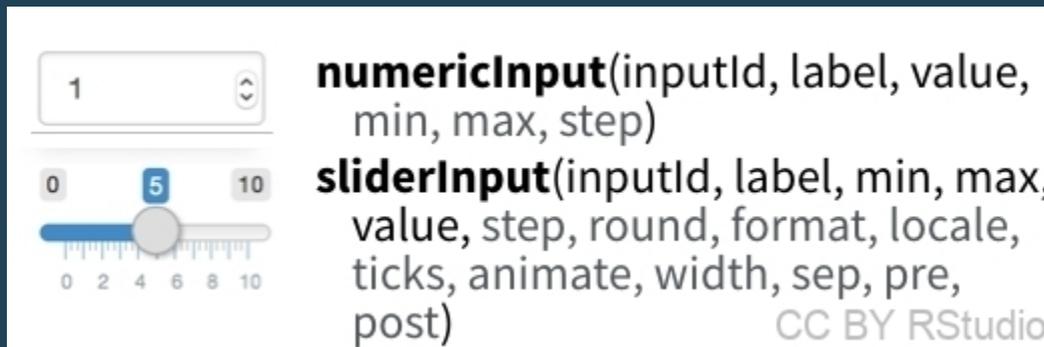
checkboxGroupInput(inputId, label, choices, selected, inline)

CC BY RStudio

Inputs - Numérique

Les *inputs* numériques permettent de sélectionner une valeur numérique parmi un ensemble prédéfini.

- **numericInput()** permet de saisir une valeur en l'entrant dans un champs
- **sliderInput()** permet de saisir une valeur en faisant défiler un *slider*
- **rangeInput()** permet de sélectionner graphiquement une étendue numérique



The image shows two examples of numeric input widgets. The top one is a `numericInput` widget, which is a text box containing the number '1' and a small arrow icon on the right. The bottom one is a `sliderInput` widget, which is a horizontal slider with a blue track and a grey knob. The track has major ticks at 0, 2, 4, 6, 8, and 10. The knob is currently positioned at the value 5. The text to the right of the widgets lists the arguments for each function: `numericInput(inputId, label, value, min, max, step)` and `sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)`. The text 'CC BY RStudio' is visible in the bottom right corner of the screenshot.

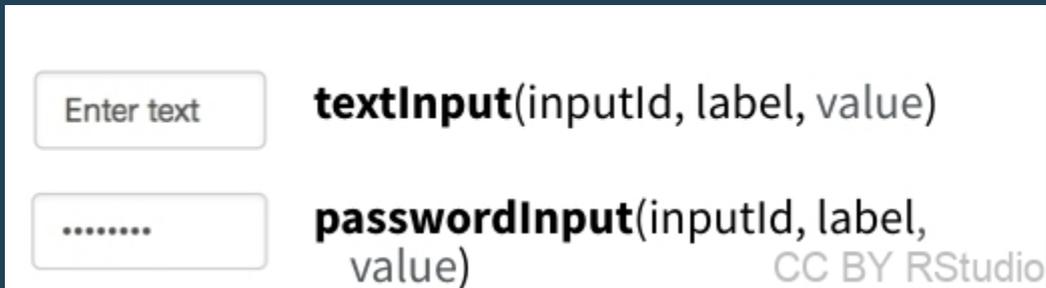
numericInput(inputId, label, value, min, max, step)

sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)

CC BY RStudio

Inputs - Texte

- `textInput()` (pour un texte court) ou `textAreaInput()` (pour un texte plus long, par exemple un paragraphe de commentaires)
- `passwordInput()` pour un mot de passe (les caractères sont masqués)



The diagram illustrates two types of text inputs and their corresponding RStudio functions. The first row shows a text input field containing the text "Enter text" next to the function `textInput(inputId, label, value)`. The second row shows a password input field containing seven dots "....." next to the function `passwordInput(inputId, label, value)`. A small watermark "CC BY RStudio" is visible in the bottom right corner of the diagram area.

Inputs - Divers



fileInput(inputId, label, multiple, accept)

checkboxInput(inputId, label, value)

dateInput(inputId, label, value, min, max, format, startview, weekstart, language)

dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)

CC BY RStudio

Parmi les autres widgets les plus utiles, on trouve:

- **fileInput()** qui permet de charger un fichier
- **checkboxInput()** qui permet de spécifier si un paramètre a pour valeur **TRUE** ou **FALSE**
- **dateInput()** et **dateRangeInput()** qui permettent de spécifier des dates.

Inputs - Récupérer les valeurs

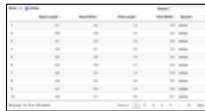
- On peut récupérer les valeurs d'un **input**, dans la partie **server** d'une appli, en indiquant son nom dans l'objet **input** — une sorte de liste —, donc avec l'opérateur **\$**: **input\$ID_DE_L_INPUT**

```
#####  
# Partie UI #  
#####  
sliderInput(inputId = "monSlider1", label = "Slider 1",  
            min = 0, max = 50, value = 25, step = 5),  
sliderInput(inputId = "monSlider2", label = "Slider 2",  
            min = -100, max = 100, value = 0, step = 10),  
selectInput(inputId = "maSelection", label = "Choisir un élément",  
            choices = c("Pomme", "Poire", "Papaye"), multiple = FALSE)  
#####  
# Partie server #  
#####  
input$monSlider1  
# > 25  
input$monSlider2  
# > 0  
input$maSelection  
# > "Pomme"
```

Outputs (1)

- Les **outputs** sont les composants de l'interface graphique qui permettent d'afficher des éléments résultant d'un traitement dans R (graphiques, tables, textes...).

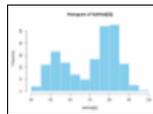
Outputs



dataTableOutput(outputId, icon, ...)



imageOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)



plotOutput(outputId, width, height, click, dblclick, hover, hoverDelay, inline, hoverDelayType, brush, clickId, hoverId)

```
"data.frame": 3 obs. of 2 variables:  
 $ Sepal.Length: num  5.1 4.9 4.7  
 $ Petal.Length : num  1.6 1.4 1.5
```

verbatimTextOutput(outputId)

	Repeat.Length	Repeat.Width	Peak.Length	Peak.Width	Species
1	5.10	3.50	5.00	0.20	setosa
2	4.90	3.20	5.20	0.20	setosa
3	4.70	3.20	5.30	0.20	setosa
4	5.40	3.10	5.10	0.20	setosa
5	5.20	3.60	5.00	0.20	setosa
6	5.20	3.40	5.10	0.20	setosa

tableOutput(outputId)

foo

textOutput(outputId, container, inline)



uiOutput(outputId, inline, container, ...)

htmlOutput(outputId, inline, container, ...)

Outputs (2)

- Les **outputs** sont les composants de l'interface graphique qui permettent d'afficher des éléments résultant d'un traitement dans R (graphiques, tables, textes...).
- Ils font partie de l'interface graphique, et se déclarent donc, comme les **inputs**, dans la partie **UI** :
 - **graphiques** : `plotOutput()`
 - **texte** : `textOutput()`
 - **table** : `tableOutput()`

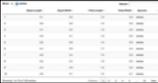
Il est également possible de produire des outputs de type

- **image** (`imageOutput()`): à ne pas confondre avec `plotOutput()` : ce sont des images qui ne font pas l'objet d'une génération par R mais sont simplement affichées, en .jpg ou .png par exemple.
- **ui** : (`uiOutput()`): cela correspond à la production d'un nouveau "morceau" d'interface utilisateur (du html, donc!).]

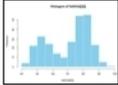
Outputs (3)

- Shiny génère (*render*) le contenu des **outputs** et, celui-ci réagit donc à du code écrit dans la partie **server** de shiny :

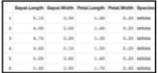
Outputs - `render*()` and `*Output()` functions work together to add R output to the UI

 **DT::renderDataTable**(`expr`, `options`, `callback`, `escape`, `env`, `quoted`) **works with** **dataTableOutput**(`outputId`, `icon`, ...)

 **renderImage**(`expr`, `env`, `quoted`, `deleteFile`) **imageOutput**(`outputId`, `width`, `height`, `click`, `dblclick`, `hover`, `hoverDelay`, `inline`, `hoverDelayType`, `brush`, `clickId`, `hoverId`)

 **renderPlot**(`expr`, `width`, `height`, `res`, ..., `env`, `quoted`, `func`) **plotOutput**(`outputId`, `width`, `height`, `click`, `dblclick`, `hover`, `hoverDelay`, `inline`, `hoverDelayType`, `brush`, `clickId`, `hoverId`)

 **renderPrint**(`expr`, `env`, `quoted`, `func`, `width`) **verbatimTextOutput**(`outputId`)

 **renderTable**(`expr`, ..., `env`, `quoted`, `func`) **tableOutput**(`outputId`)

foo **renderText**(`expr`, `env`, `quoted`, `func`) **textOutput**(`outputId`, `container`, `inline`)

 **renderUI**(`expr`, `env`, `quoted`, `func`) **uiOutput**(`outputId`, `inline`, `container`, ...) & **htmlOutput**(`outputId`, `inline`, `container`, ...)

CC BY RStudio

Remarquez, dans le graphique ci-dessus, comme une fonction **renderTruc()** côté **server** correspond à une fonction **TrucOutput()** côté UI.

Outputs (4)

- Dans la partie **ui**, on déclare des composants "graphiques" de certains types (**trucOutput()**), et on calcule le contenu à afficher dans ces composants dans la partie **server** (**renderTruc()**), en liant les deux via la déclaration d'un **output**

RENDER REACTIVE OUTPUT

```
library(shiny)
ui <- fluidPage(
  textInput("a", "", "A"),
  textOutput("b")
)

server <-
function(input, output){
  output$b <-
    renderText({
      input$a
    })
}

shinyApp(ui, server)
```

render*() functions (see front page)

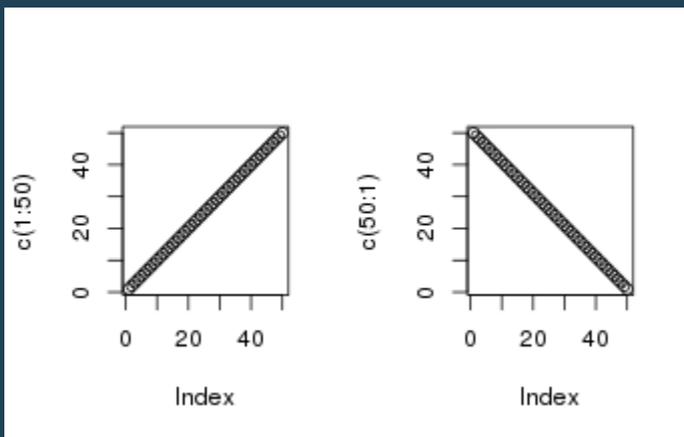
Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.

Save the results to **output\$<outputId>**

CC BY RStudio

Outputs (5)

```
#####  
# Partie UI #  
#####  
  
plotOutput(outputId = "graphique1"),  
plotOutput(outputId = "graphique2"),  
textOutput(outputId = "texte1")  
  
# [...]  
  
#####  
# Partie server #  
#####  
output$texte1 <- renderText({print(c(1:50))})  
output$graphique1 <- renderPlot({plot(1:50)})  
output$graphique2 <- renderPlot({plot(50:1)})
```



```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50  
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50  
## [47] 47 48 49 50
```

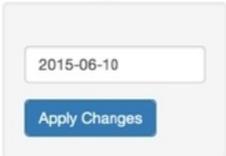
Partie 3 : Organisation de l'UI : Layouts et panels

Les Panels (1)

- Les **panels** permettent de réunir différents éléments (widgets, textes, images...). Les différents types de panels correspondent à différents **styles** (par exemple, fonds gris pour `wellPanel()`, caractères colorés pour `titlePanel()`, etc.)

Combine multiple elements into a "single element" that has its own properties with a panel function, e.g.

```
wellPanel(dateInput("a", ""),  
          submitButton()  
)
```



<code>absolutePanel()</code>	<code>navlistPanel()</code>
<code>conditionalPanel()</code>	<code>sidebarPanel()</code>
<code>fixedPanel()</code>	<code>tabPanel()</code>
<code>headerPanel()</code>	<code>tabsetPanel()</code>
<code>inputPanel()</code>	<code>titlePanel()</code>
<code>mainPanel()</code>	<code>wellPanel()</code>

CC BY RStudio

- Il est possible de **combiner différents types de panels** en les **juxtaposant** ou en les **emboîtant**, comme dans l'exemple ci-contre.

Tout ce que je vais vous montrer dans cette partie **concerne la partie UI des Shiny Apps!**

Les Panels (2)

- Shiny utilise le *framework* javascript/CSS **Bootstrap** qui définit une page comme une grille :
 - une page est divisée en colonnes (`column()`) et en lignes (`fluidRow()`)
 - la hauteur d'une ligne est celle de son plus grand élément
 - la page est divisée en **12 colonnes** : une colonne de largeur (`width`) **6** occupera donc la moitié de l'espace horizontal

```
fluidPage(  
  titlePanel("Ceci est un titlePanel", ),  
  fluidRow(column(width = 6, "Première colonne de largeur 6"),  
           column(width = 2, "Deuxième colonne de largeur 2"),  
           column(width = 4, "Troisième colonne de largeur 4")  
  ),  
  fluidRow(  
    column(width = 6, "Colonne de largeur 6"),  
    column(width = 6, "Colonne de largeur 6")  
  )  
)
```

Les Panels (2)

```
fluidPage(  
  titlePanel("Ceci est un titlePanel", ),  
  fluidRow(column(width = 6, "Première colonne de largeur 6"),  
           column(width = 2, "Deuxième colonne de largeur 2"),  
           column(width = 4, "Troisième colonne de largeur 4")  
          ),  
  fluidRow(  
    column(width = 6, "Colonne de largeur 6"),  
    column(width = 6, "Colonne de largeur 6")  
  )  
)
```

Ceci est un titlePanel

Ceci est une première colonne de
largeur 6

Colonne de largeur 6

Ceci est
une
deuxième
colonne
de
largeur 2

Colonne de largeur 6

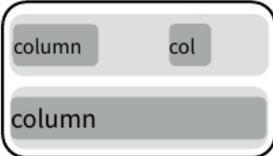
Ceci est une troisième
colonne de largeur 4

Layouts (1)

- On peut par ailleurs utiliser des types de **layouts** prédéfinis pour organiser son appli...

Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

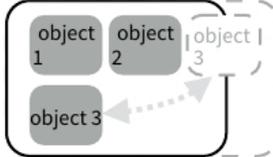
fluidRow()



The diagram shows a rectangular container divided into two rows. The top row contains two smaller rectangular boxes labeled 'column' and 'col'. The bottom row contains a single larger rectangular box labeled 'column'.

```
ui <- fluidPage(  
  fluidRow(column(width = 4),  
           column(width = 2, offset = 3)),  
  fluidRow(column(width = 12))  
)
```

flowLayout()



The diagram shows a rectangular container with a dashed border. Inside, three boxes labeled 'object 1', 'object 2', and 'object 3' are arranged in a flow. 'object 1' and 'object 2' are in the top row, and 'object 3' is in the bottom row. A dashed arrow points from 'object 3' in the bottom row back to 'object 3' in the top row, indicating a wrap-around.

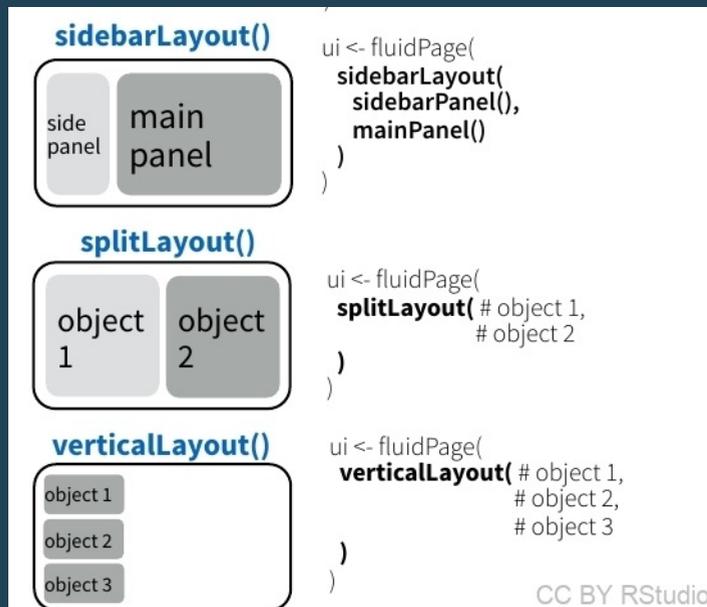
```
ui <- fluidPage(  
  flowLayout(# object 1,  
            # object 2,  
            # object 3  
)  
)
```

CC BY RStudio

- **fluidRow()** permet de définir précisément l'organisation de l'appli, en lignes et colonnes. Chaque ligne compte 12 unités de largeur au total.
- **flowLayout()** adapte le layout en fonction de la taille totale de l'élément et la taille des éléments qui le composent.
- La différence entre ces types de layout est surtout visible en modifiant la taille de la fenêtre de l'application.

Layouts (2)

- On peut par ailleurs utiliser des types de **layouts** prédéfinis pour organiser son appli...



sidebarLayout()

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

splitLayout()

```
ui <- fluidPage(
  splitLayout( # object 1,
               # object 2
  )
)
```

verticalLayout()

```
ui <- fluidPage(
  verticalLayout( # object 1,
                  # object 2,
                  # object 3
  )
)
```

CC BY RStudio

- **sidebarLayout()** divise la fenêtre en deux éléments : un élément latéral plus étroit (qui contiendra généralement des inputs) et un élément principal (qui contiendra généralement des outputs).
- **splitLayout()** permet de diviser la fenêtre en éléments de taille égale. Si le contenu des éléments est trop important l'élément devient "scrollable".
- **verticalLayout()** dispose les éléments les uns en dessous des autres.

Layouts en onglets

- Les différents *inputs* et *outputs* peuvent aussi être organisés dans des structures emboîtantes :

Layer tabPanels on top of each other,
and navigate between them, with:



```
ui <- fluidPage( tabsetPanel(  
  tabPanel("tab 1", "contents"),  
  tabPanel("tab 2", "contents"),  
  tabPanel("tab 3", "contents")))
```



```
ui <- fluidPage( navlistPanel(  
  tabPanel("tab 1", "contents"),  
  tabPanel("tab 2", "contents"),  
  tabPanel("tab 3", "contents")))
```



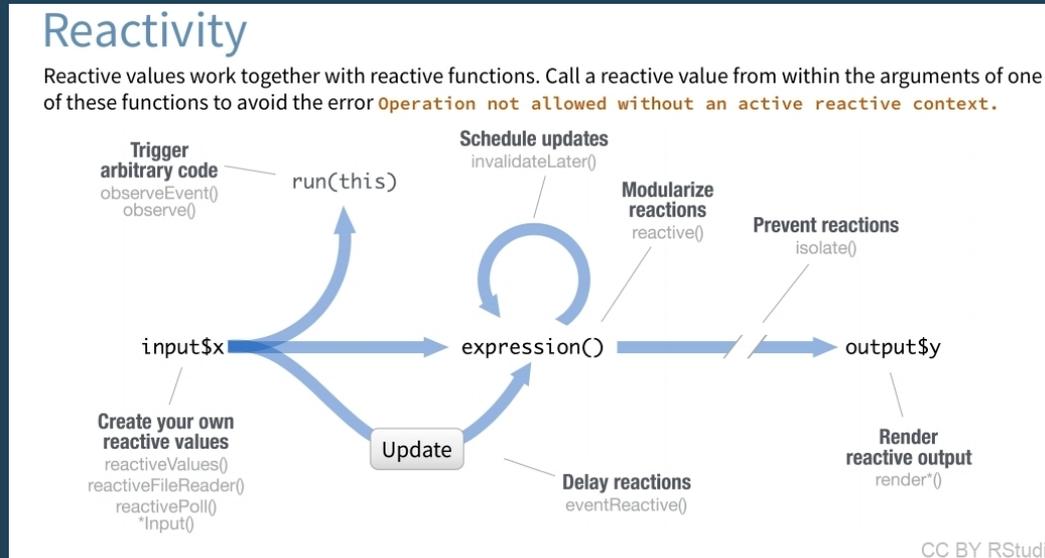
```
ui <- navbarPage(title = "Page",  
  tabPanel("tab 1", "contents"),  
  tabPanel("tab 2", "contents"),  
  tabPanel("tab 3", "contents"))
```



CC BY RStudio

Partie 4 : Réactivité

Réactivité : base

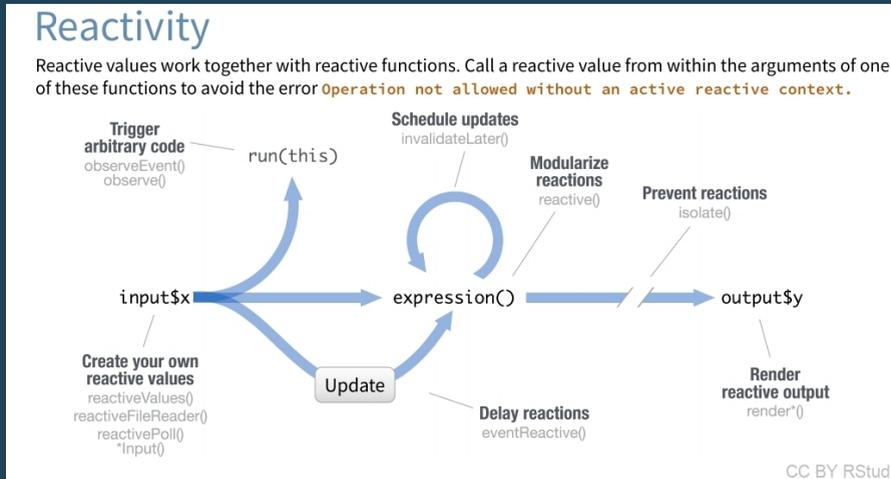


Dans sa version la plus simple, la chaîne de réactivité ressemble en fait à ceci :



Comprendre : à chaque changement de `input$x`, `expression()` est ré-évaluée et met à jour `output$y`.

Fournir des outputs réactifs



Pour rappel, voici comment l'on procède pour fournir un output réactif:

RENDER REACTIVE OUTPUT

```
library(shiny)
ui <- fluidPage(
  textInput("a", "", "A"),
  textOutput("b")
)
server <-
function(input, output){
  output$b <-
  renderText({
    input$a
  })
}
shinyApp(ui, server)
```

render*() functions

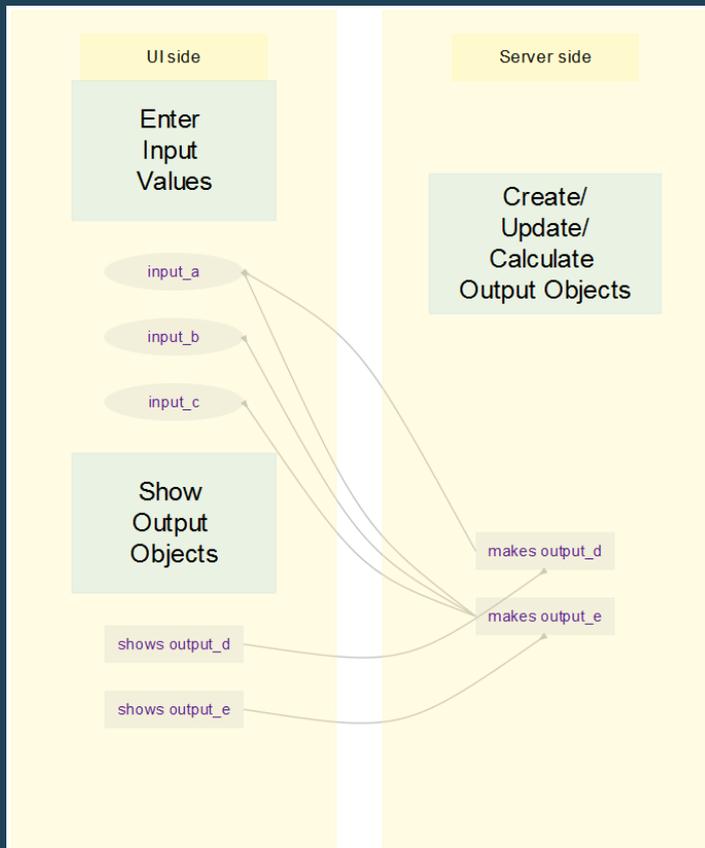
(see front page)

Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.

Save the results to **output\$<outputId>**

CC BY RStudio

Fournir des outputs réactifs - exemple (1)



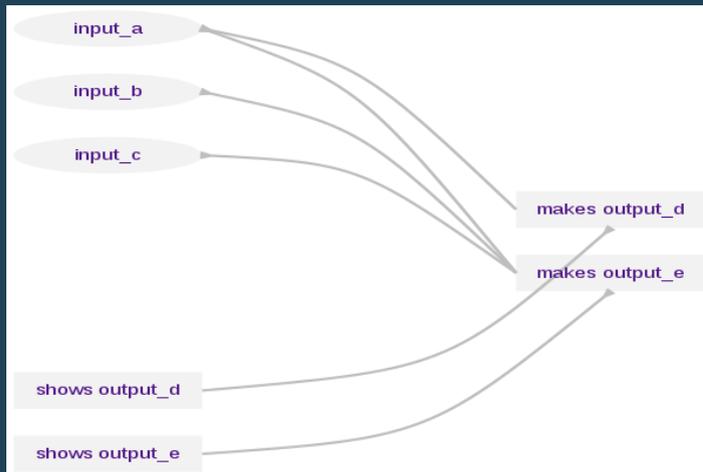
Observez le diagramme ci-contre, qui décrit un exemple comprenant :

- 3 inputs:
 - `input_a`
 - `input_b`
 - `input_c`
- 2 outputs:
 - `output_d` ; dépend de :
 - `input_a`
 - `input_b`
 - `input_c`
 - `output_e` ; dépend de :
 - `input_a`
 - `input_b`
 - `input_c`

On va utiliser cet exemple pour illustrer le fonctionnement de la **réactivité** des applications Shiny.

Fournir des outputs réactifs - exemple (2)

Commençons par simplifier et expliquer ce diagramme:



À chaque changement de valeur de **input_a** :

- La fonction **makes output_d** est exécutée et produit une sortie (**shows output_d**)
- La fonction **makes output_e** est exécutée et produit une sortie (**shows output_e**)

A chaque changement de valeur de **input_b** ou **input_c** :

- la fonction **makes output_e** est exécutée et produit une sortie (**shows output_e**)

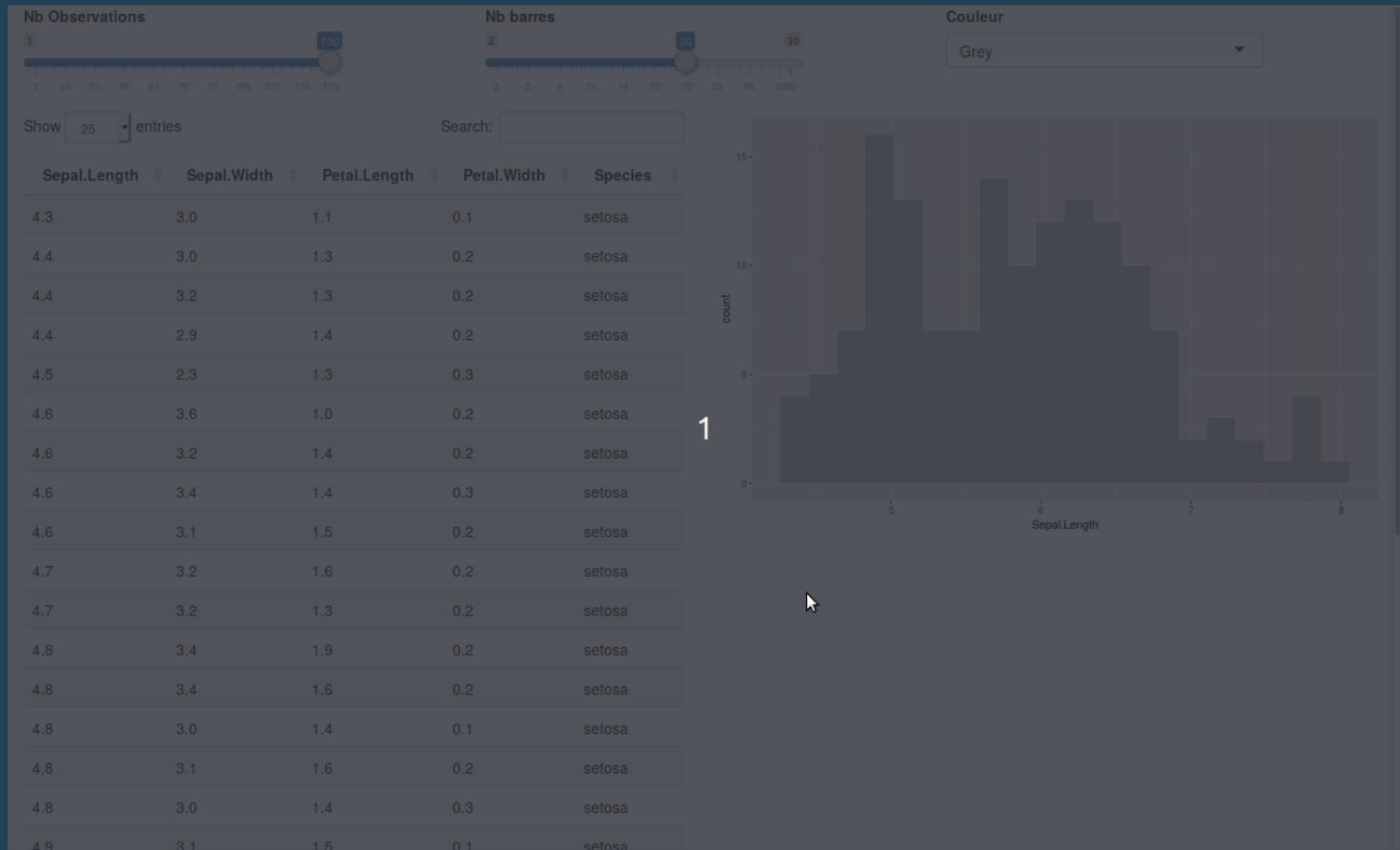
Fournir des outputs réactifs - exemple (3)

```
# Global
library(shiny)
library(tidyverse)

# UI
ui <- fluidPage(
  fluidRow(
    column(4, sliderInput(inputId = "input_a", label = "Nb Observations",
                          min = 1, max = 150, value = 150)),
    column(4, sliderInput(inputId = "input_b", label = "Nb barres",
                          min = 2, max = 30, value = 20)),
    column(4, selectInput(inputId = "input_c", label = "Couleur",
                          choices = c("Grey", "Red", "Blue", "Green")))
  ),
  fluidRow(
    column(6, dataTableOutput(outputId = "output_d")),
    column(6, plotOutput(outputId = "output_e"))
  )
)

# Server
server <- function(input, output) {
  output$output_d <- renderDataTable({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
  })
  output$output_e <- renderPlot({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
    ggplot(table_filtree) +
      geom_histogram(aes(x = Sepal.Length), bins = input$input_b,
                      fill = input$input_c)
  })
}
```

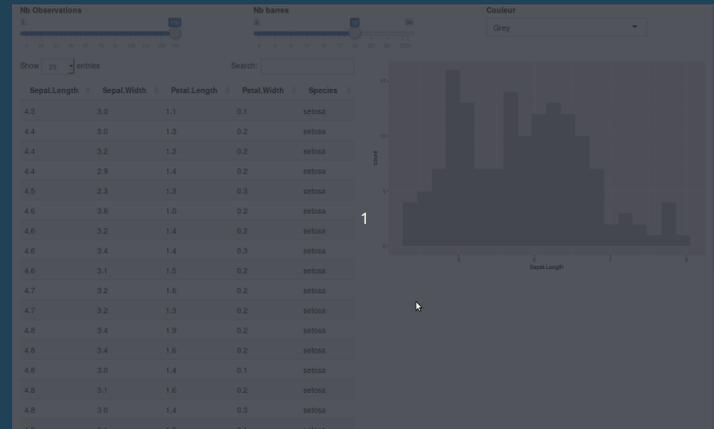
Fournir des outputs réactifs - exemple (3)



Fournir des outputs réactifs - exemple (3)

Quel est le problème avec cette application ?

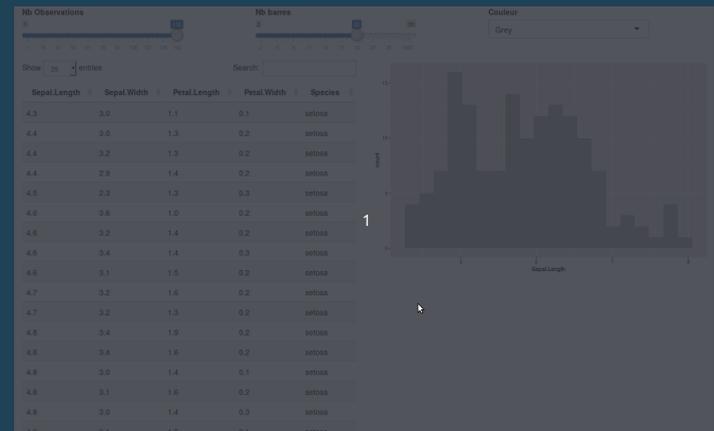
```
# Server
server <- function(input, output) {
  output$output_d <- renderDataTable({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
  })
  output$output_e <- renderPlot({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
    ggplot(table_filtree) +
      geom_histogram(aes(x = Sepal.Length),
                     bins = input$input_b,
                     fill = input$input_c)
  })
}
```



Fournir des outputs réactifs - exemple (3)

Quel est le problème avec cette application ?

```
# Server
server <- function(input, output) {
  output$output_d <- renderDataTable({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
  })
  output$output_e <- renderPlot({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
    ggplot(table_filtree) +
      geom_histogram(aes(x = Sepal.Length),
                     bins = input$input_b,
                     fill = input$input_c)
  })
}
```

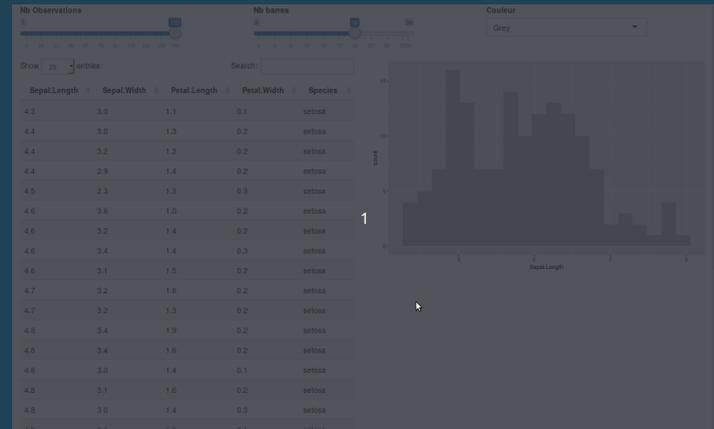


- L'histogramme et le tableau affiché ne correspondent pas aux mêmes données : l'échantillonnage est exécuté deux fois.

Fournir des outputs réactifs - exemple (3)

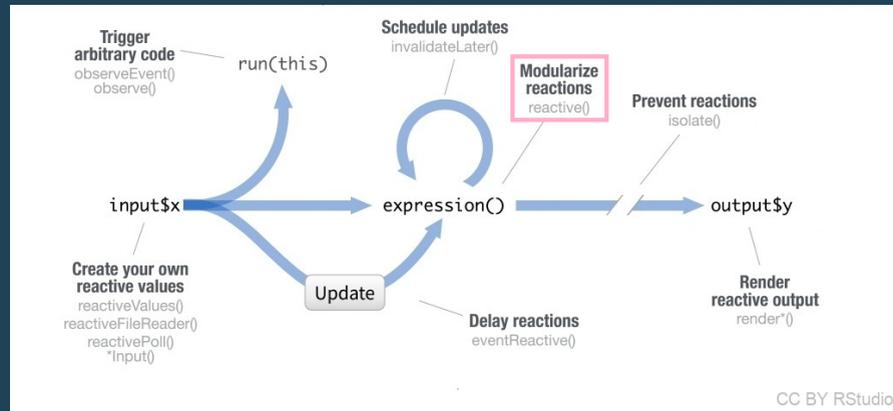
Quel est le problème avec cette application ?

```
# Server
server <- function(input, output) {
  output$output_d <- renderDataTable({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
  })
  output$output_e <- renderPlot({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
    ggplot(table_filtree) +
      geom_histogram(aes(x = Sepal.Length),
        bins = input$input_b,
        fill = input$input_c)
  })
}
```



- L'histogramme et le tableau affiché ne correspondent pas aux mêmes données : l'échantillonnage est exécuté deux fois.
- La "réactivité" ne peut reposer uniquement sur des **input**, on a aussi besoin de structures **réactives** capables de stocker (et de mettre à jour) des objets.

Modulariser les réactions



- La fonction `reactive()` permet de modulariser du code réactif...
- L'usage de `reactives` est particulièrement utile lorsque certains morceaux de code sont utilisés par **plusieurs outputs à la fois**, et permet d'éviter des redondances dans le code.

MODULARIZE REACTIONS

```
ui <- fluidPage(
  textInput("a", "", "A"),
  textInput("z", "", "Z"),
  textOutput("b")
)

server <-
function(input, output){
  re <- reactive({
    paste(input$a, input$z)})
  output$b <- renderText({
    re()
  })
}
shinyApp(ui, server)
```

reactive(x, env, quoted, label, domain)
Creates a **reactive expression** that

- **caches** its value to reduce computation
- can be called by other code
- notifies its dependencies when it has been invalidated

Call the expression with function syntax, e.g. `re()`

CC BY RStudio

Modulariser les réactions - exemple

On adapte notre code initial

```
# Server
server <- function(input, output) {
  output$output_d <- renderDataTable({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
  })
  output$output_e <- renderPlot({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
    ggplot(table_filtree) +
      geom_histogram(aes(x = Sepal.Length),
                     bins = input$input_b,
                     fill = input$input_c)
  })
}
```

Modulariser les réactions - exemple

On adapte notre code initial

```
# Server
server <- function(input, output) {
  output$output_d <- renderDataTable({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
  })
  output$output_e <- renderPlot({
    table_filtree <- iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
    ggplot(table_filtree) +
      geom_histogram(aes(x = Sepal.Length),
                     bins = input$input_b,
                     fill = input$input_c)
  })
}
```

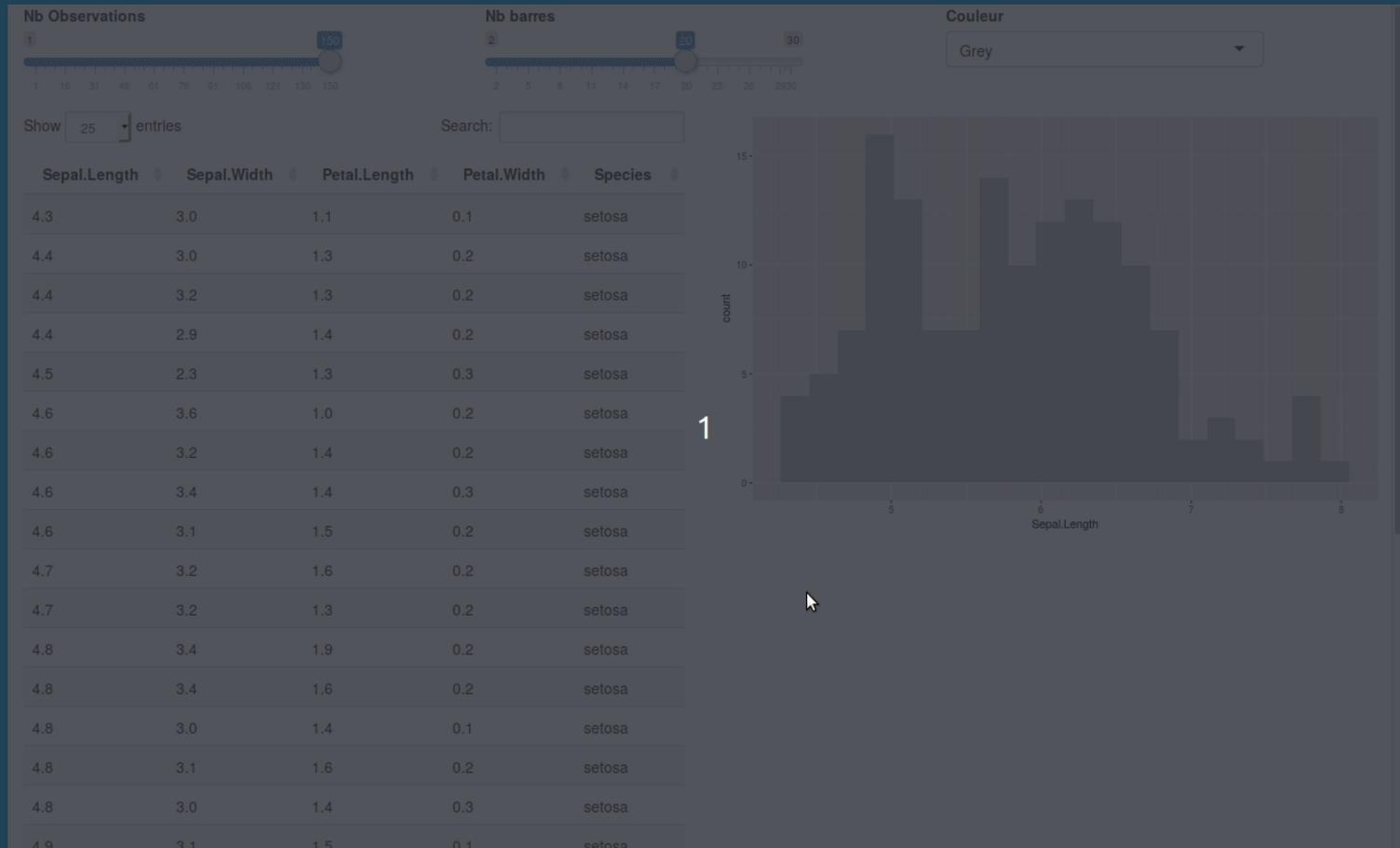
Avec une fonction `reactive()` pour stocker notre table

```
# Server
server <- function(input, output) {
  table_filtree <- reactive({
    iris %>%
      sample_n(input$input_a) %>%
      arrange(Species, Sepal.Length)
  })
  output$output_d <- renderDataTable({
    table_filtree()
  })
  output$output_e <- renderPlot({
    ggplot(table_filtree()) +
      geom_histogram(aes(x = Sepal.Length),
                     bins = input$input_b,
                     fill = input$input_c)
  })
}
```

- Une fonction `reactive()` est une **fonction** : on l'appelle donc avec des parenthèses :

```
# Declaration
abc <- reactive(mean(iris$Sepal.Length))
# Utilisation
abc()
#> 5.843333
```

Modulariser les réactions - exemple



Observer des variables (1)

- On peut parfois avoir besoin d'observer et de réagir à des changements de variables réactives, sans pour autant renvoyer un résultat.
 - Par exemple, dans l'application présentée, le *slider* du nombre de barres de l'histogramme ("Nb barres") a des bornes min et max définies dans l'**UI**, mais on peut vouloir ajuster ces bornes.
 - Quand les 150 observations sont présentes, le maximum (30) est adapté
 - Mais quand on échantillonne seulement 20 observations, le nombre de barres maximum devrait s'adapter.

On utilise pour cela la fonction **observe()** :

Observer des variables (2)

observe - Utiliser *observe* pour le code exécuté quand un *input* change, mais sans créer d'*output*.

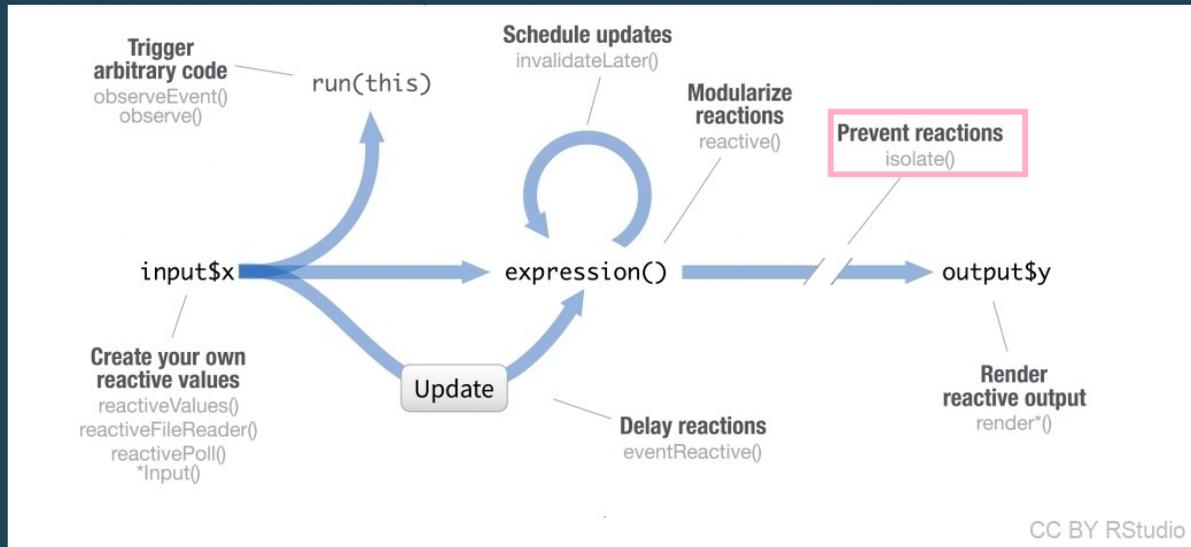


```
observe({  
  input$a  
  # code à exécuter  
})
```

```
# Server  
server <- function(input, output, session) {  
  # [...]  
  
  observe({  
    if (input$input_a > 30){  
      updateSliderInput(inputId = "input_b",  
                        session = session,  
                        min = 2,  
                        max = 30)  
    } else {  
      updateSliderInput(inputId = "input_b",  
                        session = session,  
                        min = 2,  
                        max = input$input_a,  
                        value = input$input_a/2)  
    }  
  })  
}
```

Empêcher des réactions

- On peut chercher à ne pas exécuter un code quand un élément réactif change



Empêcher des réactions

PREVENT REACTIONS

```
library(shiny)
ui <- fluidPage(
  textInput("a", "", "A"),
  textOutput("b")
)
server <-
function(input, output){
  output$b <-
  renderText({
    isolate({input$a})
  })
}
shinyApp(ui, server)
```

isolate(expr)

Runs a code block.
Returns a **non-reactive**
copy of the results.

CC BY RStudio

```
# UI
fluidRow(
  column(width = 4, textInput("word1",
                              "First word",
                              "Hello")),
  column(width = 4, textInput("word2",
                              "Second word",
                              "Master")),
  column(width = 4, textOutput("combi"))
)
# Server
output$combi <- renderText({
  paste(input$word1, isolate(input$word2))
})
```

- Ici, l'output n'est actualisé que quand le deuxième input est modifié.
- Une modification du premier input ne déclenche pas le code contenu dans le contexte réactif.

Récapitulatif des éléments réactifs

4. Réactivité (Quand un *input* change, le serveur va reconstruire chaque *output* qui en dépend(même quand la dépendance est indirecte) Ce comportement est maîtrisé par l'ajustement de la chaîne de dépendance.

RStudio® and Shiny™ are trademarks of RStudio, Inc.
[CC BY](https://creativecommons.org/licenses/by/4.0/) RStudio info@rstudio.com
844-448-1212 rstudio.com

Traduit par Asma Balti & Vincent Guyader • <http://thinkr.fr>

render* - Un *output* sera automatiquement mis à jour quand un *input* de sa fonction *render** change.



```
output$z <- renderText({
  input$a
})
```

Expression réactive - Utiliser *reactive* pour créer des objets à utiliser dans des multiples *outputs*.



```
x <- reactive({
  input$a
})

output$y <- renderText({
  x()
})

output$z <- renderText({
  x()
})
```

isolate - Utiliser *isolate* pour utiliser des *input* sans qu'il y ait de dépendance. Shiny ne reconstruit pas l'*output* quand l'*input* isolé change.



```
output$z <- renderText({
  paste(
    isolate(input$a),
    input$b
  )
})
```

observe - Utiliser *observe* pour le code exécuté quand un *input* change, mais sans créer d'*output*.



```
observe({
  input$a
  # code à exécuter
})
```

Une petite démonstration

- On va adapter l'application de base aux données "Dans Ma Rue" pour créer une application d'exploration interactive de ces données.
- On souhaite donc avoir :
 - Une carte de localisation des incidents
 - Un moyen de sélectionner l'année
 - Un moyen d'isoler des types d'incidents
 - Un graphique récapitulatif des incidents sélectionnés

Exercice

Depuis la base applicative créée auparavant, améliorer l'application pour la rendre plus adaptée à l'exploration des données.

Quelques idées/pistes :

- Remplacer la carte statique par une carte dynamique
- Intégrer une exploration des sous-types
- Intégrer les données IRIS pour les rendre elles aussi explorables
- Remplacer le graphique récapitulatif par un graphique interactif
- Changer la manière de sélectionner l'année