SHINY
PROG STAT 2

IUT Aurillac

05-01-2025

Qu’est ce que Shiny?

- Une application (app) Shiny est une application Web de R.
- C’est un outil interactif pour vos analyses, dédié à des utilisateurs non techniques.
- Grande flexibilité pour faire des dashboards ou des prototypes rapidement et avec un design professionnel
- Aucune connaissance HTML, CSS ou JavaScript n'est requise mais quelques bases HTML peuvent être un plus.
- Un serveur Shiny est requis pour héberger une application Shiny pour le monde sinon il faut avoir R/Shiny et accés au code en local pour avoir accès à l’application Shiny.
- Ce package est créé par **Joe Cheng,** il est gratuit et open source

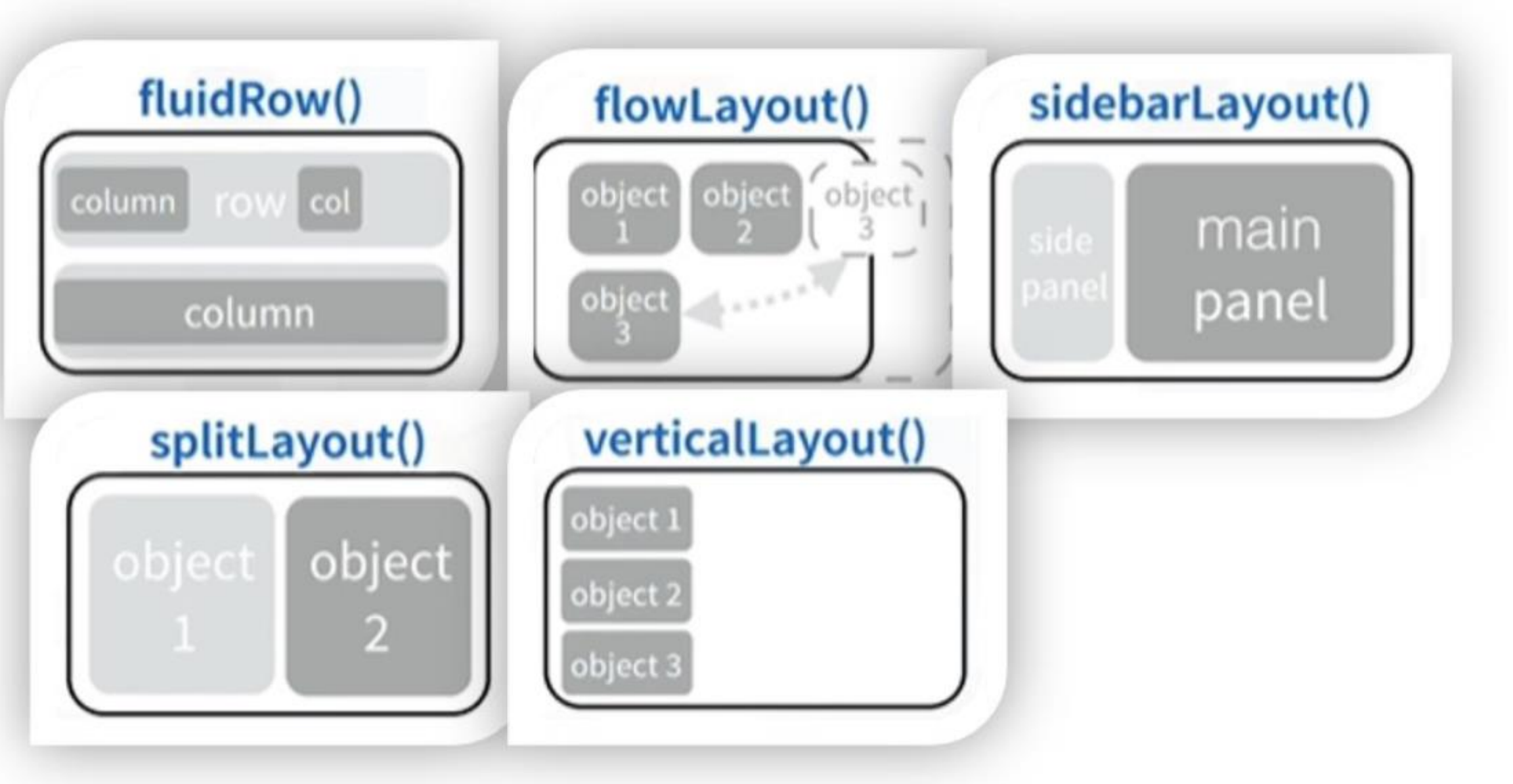
Un projet Shiny est composé de 3 composantes :

- ui.R (user interface) pour la programmation de l'interface utilisateur. Ce fichier R (ou cet objet d'interface R) contrôle le « look » et le design de l'application Shiny
- server.R est une fonction qui contrôle ce que fait l'application
- Une fonction shinyApp() ou runApp() qui lance l'application shiny. On peut aussi appuyer sur le bouton "Run App" dans RStudio

```
install.packages("shiny")
```

```
library(shiny)
runExample("01_hello")
```

- See `?builder` for more details.


fluidRow()
column
row
col
column
flowLayout()
object 1
object 2
object 3
object 3
sidebarLayout()
side panel
main panel
splitLayout()
object 1
object 2
verticalLayout()
object 1
object 2
object 3

DEMO

Shiny: des Inputs, des outputs, des reactives

Inputs	Output	Reactive
• Sont définis par l'utilisateur du Shiny comme…. • Curseur • Bouton • Case à cocher • Texte • Menu déroulant • etc	• Prend les inputs définis par l'utilisateur et les transforme • Renvoie le résultat à l'interface utilisateur pour sa visualisation tel quel	• Les expressions réactives permettent de contrôler quelles parties de votre app sont mises à jour et à quel moment • Elle évite des calculs inutiles susceptibles de ralentir votre application • Elle permette d'éviter les répétitions inutiles de code

Shiny for R : : **CHEATSHEET**

Building an App

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).

Save your template as **app.R**. Keep your app in a directory along with optional extra files.

To generate the template, type **shinyapp** and press **Tab** in the RStudio IDE or go to **File > New Project > New Directory > Shiny Application**

In **ui** nest R functions to build an HTML interface

Customize the UI with **Layout Functions**

Add Inputs with ***Input()** functions

Add Outputs with ***Output()** functions

Tell the **server** how to render outputs and respond to inputs with R

Wrap code in **render*()** functions before saving to output

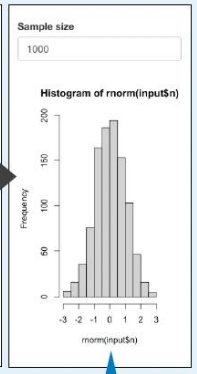
Refer to UI inputs with **input$<id>** and outputs with **output$<id>**

```
# app.R
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

server <- function(input, output, session) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

shinyApp(ui = ui, server = server)
```



Call **shinyApp()** to combine **ui** and **server** into an interactive app!

```
app-name
  app.R
  DESCRIPTION
  README
  R/
  www/
```

The directory name is the app name

(optional) used in showcase mode

(optional) directory of supplemental .R files that are sourced automatically, must be named "**R**"

(optional) directory of files to share with web browsers (images, CSS, .js, etc.), must be named "**www**"

Launch apps stored in a directory with **runApp(**<path to directory>**)**.

See annotated examples of Shiny apps by running **runExample(**<example name>**)**. Run **runExample()** with no arguments for a list of example names.

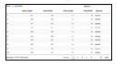
Share

Share your app in three ways:

1. **Host it on shinyapps.io**, a cloud based service from Posit. To deploy Shiny apps:
   - Create a free or professional account at **shinyapps.io**
   - Click the Publish icon in RStudio IDE, or run: **rsconnect::deployApp("**<path to directory>**")**
2. **Purchase Posit Connect**, a publishing platform for R and Python. **posit.co/products/enterprise/connect/**
3. **Build your own Shiny Server** **posit.co/products/open-source/shinyserver/**

Outputs

render*() and *Output() functions work together to add R output to the UI.

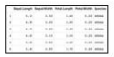


DT::**renderDataTable(**expr, options, searchDelay, callback, escape, env, quoted, outputArgs**)** — **dataTableOutput(**outputId**)**

renderImage(expr, env, quoted, deleteFile, outputArgs**)** — **imageOutput(**outputId, width, height, click, dblclick, hover, brush, inline**)**

renderPlot(expr, width, height, res, …, alt, env, quoted, execOnResize, outputArgs**)** — **plotOutput(**outputId, width, height, click, dblclick, hover, brush, inline**)**

renderPrint(expr, env, quoted, width, outputArgs**)** — **verbatimTextOutput(**outputId, placeholder**)**



renderTable(expr, striped, hover, bordered, spacing, width, align, rownames, colnames, digits, na, …, env, quoted, outputArgs**)** — **tableOutput(**outputId**)**

foo

renderText(expr, env, quoted, outputArgs, sep**)** — **textOutput(**outputId, container, inline**)**

renderUI(expr, env, quoted, outputArgs**)** — **uiOutput(**outputId, inline, container, …**)** **htmlOutput(**outputId, inline, container, …**)**

These are the core output types. See **htmlwidgets.org** for many more options.

Inputs

Collect values from the user.

Access the current value of an input object with **input$<inputId>**. Input values are **reactive**.

Action — **actionButton(**inputId, label, icon, width, …**)**

Link — **actionLink(**inputId, label, icon, …**)**

Choice 1 / Choice 2 / Choice 3 — **checkboxGroupInput(**inputId, label, **choices,** selected, inline, width, choiceNames, choiceValues**)**

Check me — **checkboxInput(**inputId, label, value, width**)**

dateInput(inputId, label, value, min, max, format, startview, weekstart, language, width, autoclose, datesdisabled, daysofweekdisabled**)**

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

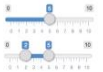
Choose File — **fileInput(**inputId, label, multiple, accept, width, buttonLabel, placeholder**)**

1 — **numericInput(**inputId, label, value, min, max, step, width**)**

•••••••• — **passwordInput(**inputId, label, value, width, placeholder**)**

Choice A / Choice B / Choice C — **radioButtons(**inputId, label, **choices,** selected, inline, width, choiceNames, choiceValues**)**

Choice 1 / Choice 1 / Choice 2 — **selectInput(**inputId, label, choices, selected, multiple, selectize, width, size**)** Also **selectizeInput()**

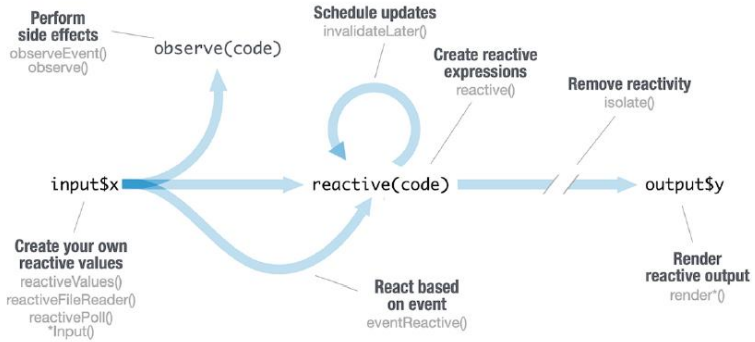


sliderInput(inputId, label, min, max, **value,** step, round, format, locale, ticks, animate, width, sep, pre, post, timeFormat, timezone, dragRange**)**

Enter text — **textInput(**inputId, label, value, width, placeholder**)** Also **textAreaInput()**

Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error **Operation not allowed without an active reactive context.**



CREATE YOUR OWN REACTIVE VALUES

```
# *Input() example
ui <- fluidPage(
 textInput("a","","A")
)
```

***Input() functions**
Each input function creates a reactive value stored as **input$<inputId>**.

```
#reactiveVal example
server <- function(input,output){
 rv <- reactiveVal()
 rv$number <- 5
}
```

reactiveVal(...)
Creates a single reactive values object.

reactiveValues(...)
Creates a list of names reactive values.

CREATE REACTIVE EXPRESSIONS

```
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)

Reactive expressions:
- **cache** their value to reduce computation
- can be called elsewhere
- notify dependencies when invalidated

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

REACT BASED ON EVENT

```
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go"),
 textOutput("b")
)

server <- function(input,output){
 re <- eventReactive(
  input$go,{input$a}
 )
 output$b <- renderText({
  re()
 })
}
```

eventReactive(eventExpr, valueExpr, event.env, event.quoted, value.env, value.quoted, ..., label, domain, ignoreNULL, ignoreInit)
Creates reactive expression with code in 2nd argument that only invalidates when reactive values in 1st argument change.

RENDER REACTIVE OUTPUT

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

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

shinyApp(ui, server)
```

render*() functions
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>**.

PERFORM SIDE EFFECTS

```
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go")
)

server <- function(input,output){
 observeEvent(
  input$go,{
  print(input$a)
  }
 )
}

shinyApp(ui, server)
```

observe(x, env)
Creates an observer from the given expression.

observeEvent(eventExpr, handlerExpr, event.env, event.quoted, handler.env, handler.quoted, ..., label, suspended, priority, domain, autoDestroy, ignoreNULL, ignoreInit, once)
Runs code in 2nd argument when reactive values in 1st argument change.

REMOVE REACTIVITY

```
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.

UI - An app's UI is an HTML document.

Use Shiny's functions to assemble this HTML with R.

```
fluidPage(
  textInput("a","")
)
## <div class="container-fluid">
##   <div class="form-group shiny-input-container">
##     <label for="a"></label>
##     <input id="a" type="text"
##       class="form-control" value=""/>
##   </div>
## </div>
```

Returns HTML

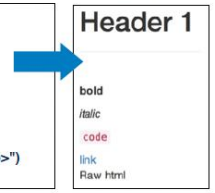
HTML Add static HTML elements with **tags**, a list of functions that parallel common HTML tags, e.g. **tags$a()**. Unnamed arguments will be passed into the tag; named arguments will become tag attributes.

Run **names(tags)** for a complete list.

```
tags$h1("Header") -> <h1>Header</h1>
```

The most common tags have wrapper functions. You do not need to prefix their names with **tags$**

```
ui <- fluidPage(
 h1("Header 1"),
 hr(),
 br(),
 p(strong("bold")),
 p(em("italic")),
 p(code("code")),
 a(href="", "link"),
 HTML("<p>Raw html</p>")
)
```



CSS To include a CSS file, use **includeCSS()**, or
1. Place the file in the **www** subdirectory
2. Link to it with:

```
tags$head(tags$link(rel = "stylesheet",
 type = "text/css", href = "<file name>"))
```

JS To include JavaScript, use **includeScript()** or
1. Place the file in the **www** subdirectory
2. Link to it with:

```
tags$head(tags$script(src = "<file name>"))
```

IMAGES To include an image:
1. Place the file in the **www** subdirectory
2. Link to it with img(src="<file name>")

Layouts

Use the **bslib** package to lay out the your app and its components.

PAGE LAYOUTS

Dashboard layouts

page_sidebar() A sidebar page
page_navbar() Multi-page app with a top navigation bar
page_fillable() A screen-filling page layout

Basic layouts

page() **page_fluid()** **page_fixed()**

USER INTERFACE LAYOUTS

Multiple columns

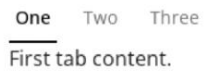
layout_columns() Organize UI elements into Bootstrap's 12-column CSS grid
layout_column_wrap() Organize elements into a grid of equal-width columns

Multiple panels

navset_tab()

navset_pill()

navset_underline()



nav_panel() Content to display when given item is selected
nav_menu() Create a menu of nav items
nav_item() Place arbitrary content in the nav panel
nav_spacer() Add spacing between nav items

Also dynamically update nav containers with **nav_select()**, **nav_insert()**, **nav_remove()**, **nav_show()**, **nav_hide()**.

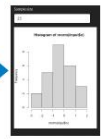
Sidebar layout

sidebar() **layout_sidebar()** **toggle_sidebar()**

Themes

Use the **bslib** package to add existing themes to your Shiny app ui, or make your own.

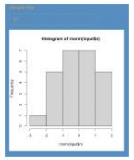
```
library(bslib)
ui <- fluidPage(
  theme = bs_theme(
    bootswatch = "darkly",
    ...
  )
)
```



bootswatch_themes() Get a list of themes.

Build your own theme by customizing individual arguments.

bs_theme(bg = "#558AC5",
fg = "#F9B02D",
...**)**



?bs_theme for a full list of arguments.

bs_themer() Place within the server function to use the interactive theming widget.

```
server <- function(input, output, session) {
  string <- reactive(paste0("Hello ", input$name, "!"))
  output$greeting <- renderText(string())
}
```

LES EXPRESSIONS REACTIVES

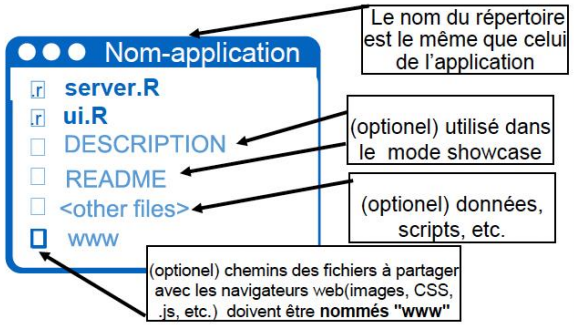
Shiny Aide-mémoire

Plus d'infos sur shiny.rstudio.com

Shiny 0.10.0 Updated: 6/14

RStudio

1. Structure Chaque application est un répertoire contenant un fichier server.R et un fichier ui.R (et éventuellement des fichiers facultatifs)



Les fonctions *render**

fonction	prend	crée
renderDataTable	tout objet équivalent à un tableau	dataTables.js table
renderImage	liste d'attributs d'image	image HTML
renderPlot	plot	plot
renderPrint	tout output imprimé	texte
renderTable	tout objet équivalent à un tableau	table
renderText	chaîne de caractères	texte
renderUI	objet Shiny tag ou HTML	élément UI(HTML)

Les valeurs d'*input* réactives doivent être utilisées dans :

render* - crée une composante UI
reactive - crée une expression réactive
observe - crée un ***reactive observer***
isolate - crée une copie non réactive d'un objet réactif

2. server.R est un ensemble d'instructions permettant de construire les éléments de l'application Shiny. Pour écrire le server.R:

A. Ecrire server.R avec le minimum de code nécessaire, **shinyServer(function(input, output) {})**

B. Définir les composantes R de l'application entre les accolades qui suivent **function(input, output)**

C. Enregistrer chaque composante R dans l'UI comme **output$<nom composante>**

D. Créer chaque composante *output* avec une fonction *render**

E. Donner à chaque fonction *render** le code R nécessaire au *Server* pour construire la composante. Le serveur va reconnaître chaque valeur réactive qui apparaît dans le code et va la reconstruire chaque fois que sa valeur change

F. Faire référence aux valeurs des widgets avec **input$<nom widget>**

server.R

```
# charger les librairies, scripts, et données

A shinyServer(function(input, output) { B

# définir des variables spécifiques à l'utilisateur

  output$text <- renderText({
    input$title
  })

C output$plot <- renderPlot({ D
    x <- mtcars[ , input$x] F
E   y <- mtcars[ , input$y]
    plot(x, y, pch = 16)
  })

})
```

3. Exécution Placer le code où il sera exécuté le minimum nécessaire de fois

Exécuté une seule fois- Le code placé à l'extérieur du ***shinyServer*** s'exécute une seule fois, lors du 1er lancement de l'*app*. Utiliser ce code pour mettre en place les éléments dont le serveur n'a besoin qu'une seule fois.

Exécuté une seule fois par utilisateur- Le code placé dans le ***shinyServer*** va être exécuté chaque fois qu'un utilisateur lance l'*app* (ou rafraîchit son navigateur). Utiliser ce code pour mettre en place les éléments qui ne seront nécessaires qu'une fois pour chaque utilisateur,

Exécuté souvent- Le code placé dans une fonction *render*, reactive*, ou *observe* va être exécuté plusieurs fois. Placer ici uniquement le code dont le serveur a besoin pour reconstruire une composante UI après la modification d'un widget.

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.

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 executer
})
```


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

ui.R

```
A shinyUI(fluidPage(

    titlePanel("Données mtcars"),
    sidebarLayout(
  B sidebarPanel(

    C textInput("titre", "titre du graphique:",
        value = "x v y"),

      selectInput("x", "Choisissez une variable x:",
        choices = names(mtcars),
        selected = "disp"),

      selectInput("y", " Choisissez une variable y:",
        choices = names(mtcars),
        selected = "mpg")
    ),

    mainPanel(
      h3(textOutput("text")),
      plotOutput("plot")
    )
  )
))
```

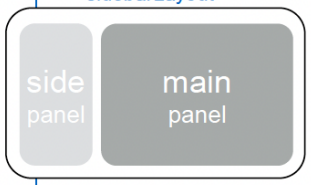
5. ui.R Une description de l'interface utilisateur (UI) de votre *app*, la page web qui affiche votre *app*. Pour écrire ui.R:

A Fournir le minimum nécessaire pour ui.R, shinyUI(fluidPage())

* note: utiliser **navbarPage** au lieu de **fluidPage** si l'*app* doit comprendre plusieurs pages connectées par une barre de navigation

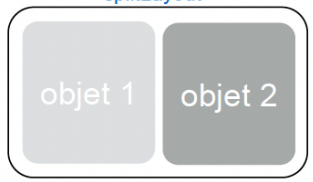
B Construire un squelette pour l'UI. SidebarLayout fournit un squelette par défaut lorsqu'il est utilisé avec sidebarPanel et mainPanel. SplitLayout, flowLayout, et inputLayout divisent la page en régions équi-espacées. FluidRow et column fonctionnent de pair pour créer un squelette en grille, utilisable pour présenter une page ou un panneau.

sidebarLayout



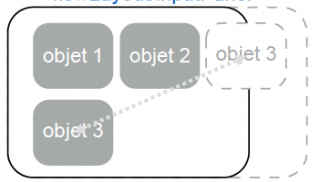

```
shinyUI(fluidPage(
  sidebarLayout(
    sidebarPanel(...),
    mainPanel(...)
  )
))
```

splitLayout



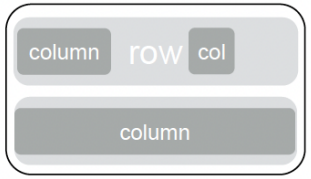

```
shinyUI(fluidPage(
  splitLayout(
    numericInput(...),
    selectInput(...)
  )
))
```

flowLayout/inputPanel




```
shinyUI(fluidPage(
  flowLayout(
    numericInput(...),
    selectInput(...),
    sliderInput(...)
  )
))
```

fluidRow




```
shinyUI(fluidPage(
  fluidRow(
    column(width = 4, ...),
    column(width = 2,
      offset = 3, ...),
  ),
  fluidRow(
    column(width = 12, ...)
  )
))
```

C Dans chaque panneau ou colonne, placer:

Les composantes R - Ce sont les objets *output* définis dans server.R. Pour placer une composante:

1. Choisir une fonction ***Output*** qui construit le type d'objet à placer dans UI.
2. Passer à la fonction ***Output*** une chaîne de caractères qui correspond au nom de l'object dans server.R, exemple:

output$plot <- renderPlot({ ... }) ⟷ plotOutput("plot")

fonctions *Outputs*

dataTableOutput	tableOutput
htmlOutput	textOutput
imageOutput	uiOutput
plotOutput	verbatimTextOutput

Widgets - Le 1er argument de chaque fonction widget est le nom pour le widget. Utiliser input$<nom> pour accéder à la valeur courante du widget dans server.R

widget	fonctions	arguments courants
Action button	actionButton	inputId, label
checkbox	checkboxInput	inputId, label, value
checkbox group	checkboxGroupInput	inputId, label, choices, selected
date selector	dateInput	inputId, label, value, min, max, format
date range selector	dateRangeInput	inputId, label, start, end, min, max, format
file uploader	fileInput	inputId, label, multiple
Number field	numericInput	inputId, label, value, min, max, step
Radio buttons	radioButtons	inputId, label, choices, selected
select box	selectInput	inputId, label, choices, selected, multiple
slider	sliderInput	inputId, label, min, max, value, step
submit button	submitButton	text
text field	textInput	inputId, label, value

Eléments HTML- Ajout des éléments html avec les fonctions Shiny similaires aux tags HTML.

a	tags$col	tags$form	tags$input	tags$output	tags$sub
tags$abbr	tags$colgroup	h1	tags$ins	p	tags$summary
tags$address	tags$command	h2	tags$kbd	tags$param	tags$sup
tags$area	tags$data	h3	tags$keygen	pre	tags$table
tags$article	tags$datalist	h4	tags$label	tags$progress	tags$tbody
tags$aside	tags$dd	h5	tags$legend	tags$q	tags$td
tags$audio	tags$del	h6	tags$li	tags$ruby	tags$textarea
tags$b	tags$details	tags$head	tags$link	tags$rp	tags$tfoot
tags$base	tags$dfn	tags$header	tags$mark	tags$rt	tags$th
tags$bdi	div	tags$hgroup	tags$map	tags$s	tags$thead
tags$bdo	tags$dl	hr	tags$menu	tags$samp	tags$time
tags$blockquote	tags$dt	HTML	tags$meta	tags$script	tags$title
br	em	tags$i	tags$meter	tags$section	tags$tr
tags$body	tags$embed	tags$iframe	tags$nav	tags$select	tags$track
tags$button	tags$eventsource	img	tags$noscript	tags$small	tags$u
tags$canvas	tags$fieldset	includeCSS	tags$object	tags$source	tags$ul
tags$caption	tags$figcaption	includeMarkd	tags$ol	span	tags$var
tags$cite	tags$figure	own	tags$optgroup	strong	tags$video
code	tags$footer	includeScript	tags$option	tags$style	tags$wbr

6. Exécuter votre *app*

runApp - exécuter en local
runGitHub - exécuter depuis des fichiers hébergés sur www.GitHub.com
runGist - exécuter depuis des fichiers enregistrés comme un gist (gist.github.com)
runURL - exécuter depuis des fichiers enregistrés dans des URL

7. Partager votre *app*

Lancer votre *app* comme une page web dynamique que les utilisateurs peuvent consulter en ligne

ShinyApps.io

Héberger votre *app* sur le serveur RStudio. Options gratuites et payantes.
www.shinyapps.io

Shiny Server

Construire un serveur linux pour héberger votre *app*.Gratuit et open source.
shiny.rstudio.com/deploy

Shiny Server Pro

Construire un serveur commercial avec authentification, gestion des ressources, et plus.
shiny.rstudio.com/deploy

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

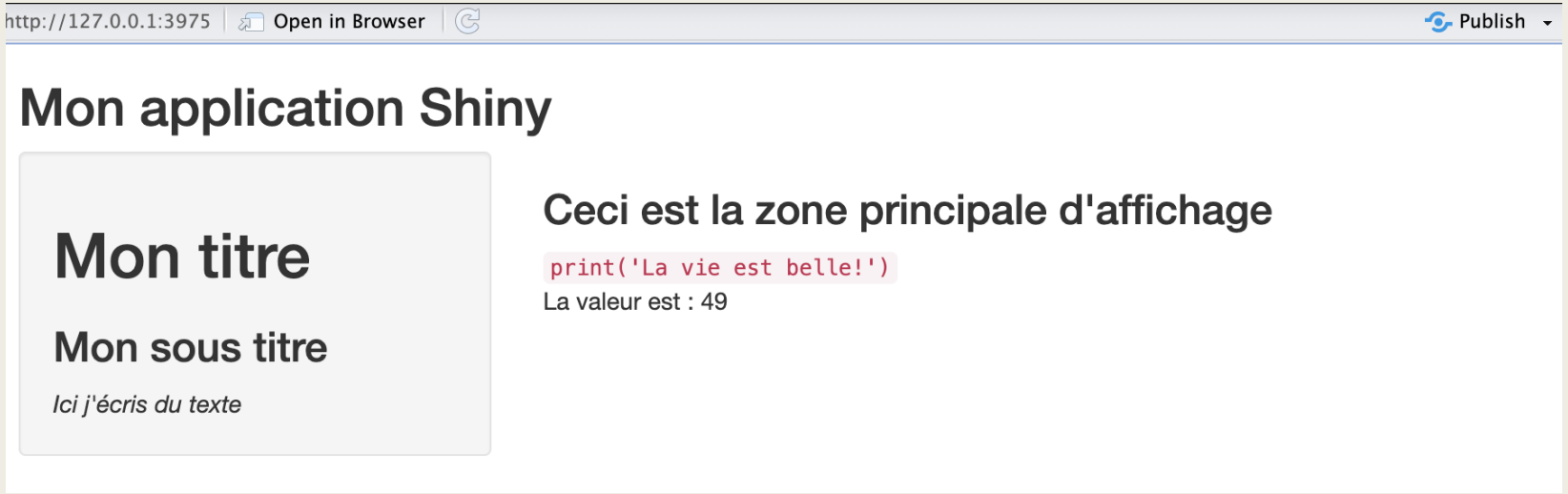
Exercice 1

- Télécharger le dossier « intro_01” et runner cette application shiny dans R
- Maintenant créez avec cette app, une application qui salue l'utilisateur par son nom. Vous avez besoin pour faire cela de certaines fonctions Shiny. J'ai donc inclus ci-dessous quelques lignes de code. Pensez aux lignes que vous utiliserez, puis copiez-les au bon endroit dans une application Shiny.

```
tableOutput("mortgage")
output$salutation <- renderText({
  paste0("Bonjour ", input$nom)
})
numericInput("age", "How old are you?", value = NA)
textInput("nom", "Quel est ton nom ?")
textOutput("salutation")
output$histogram <- renderPlot({
  hist(rnorm(1000))
}, res = 96)
```

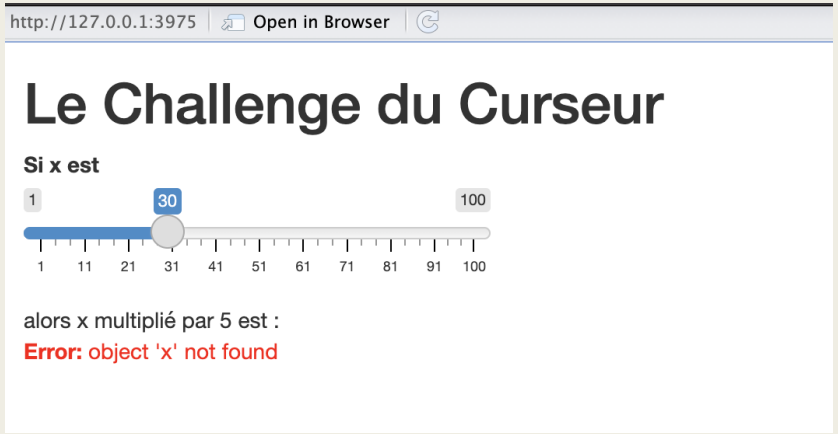
Exercice 2

- Télécharger le dossier « intro_02” et runner cette application shiny dans R
- Maintenant avec cette app, faite une application qui affiche un nombre au hasard dès que le shiny démarre. Ce nombre est un entier compris entre 1 et 100. Utiliser la solution de l’exercice précédent comme inspiration. Le résultat devra ressembler à quelque chose de similaire à la copie d’écran ci-dessous:



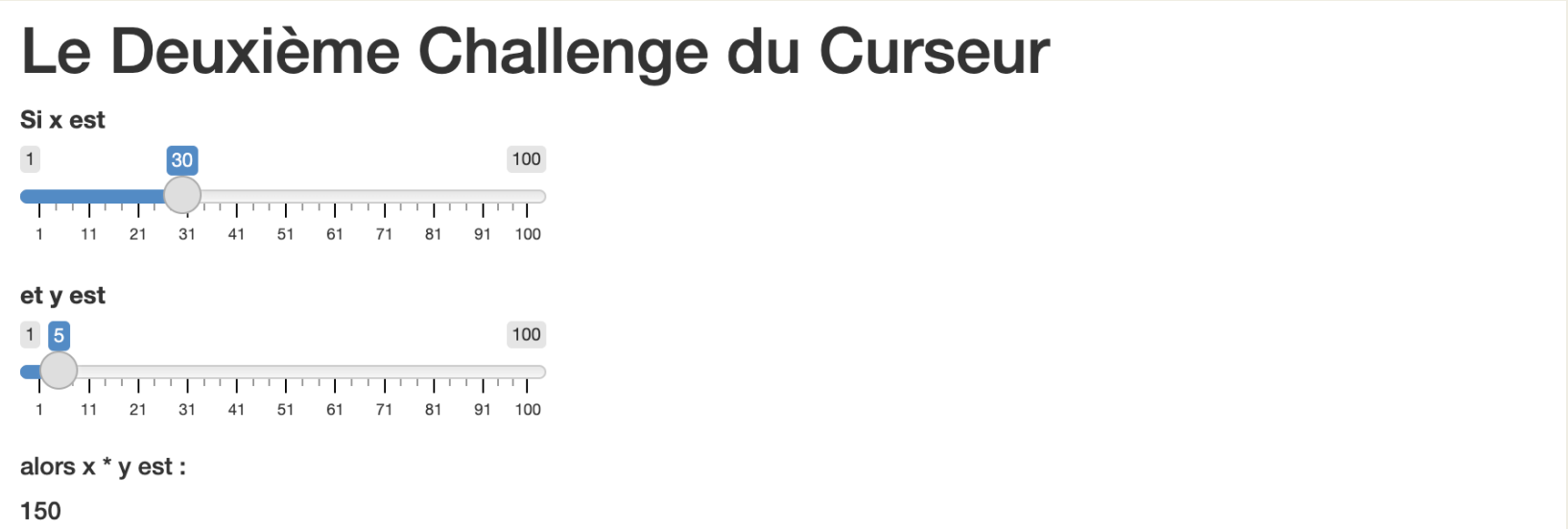
Exercice 3

- Maintenant supposons que votre ami.e souhaite concevoir une application permettant à l'utilisateur de définir un nombre (x) compris entre 1 et 100 et affichant le résultat de la multiplication de ce nombre par 5. Il s'agit de sa première tentative.
- Télécharger le dossier « intro_03” pour visualiser le résultat de cette tentative.
- Malheureusement, l’application shiny ne marche pas. Pouvez-vous aider votre ami.e à trouver et corriger le bug ?



Exercice 4:

- Étendez l'application de l'exercice précédent pour permettre à l'utilisateur de définir la valeur du multiplicateur, y, afin que l'application produise la valeur de x * y. Le résultat final devrait ressembler à ceci :



Exercice 5:

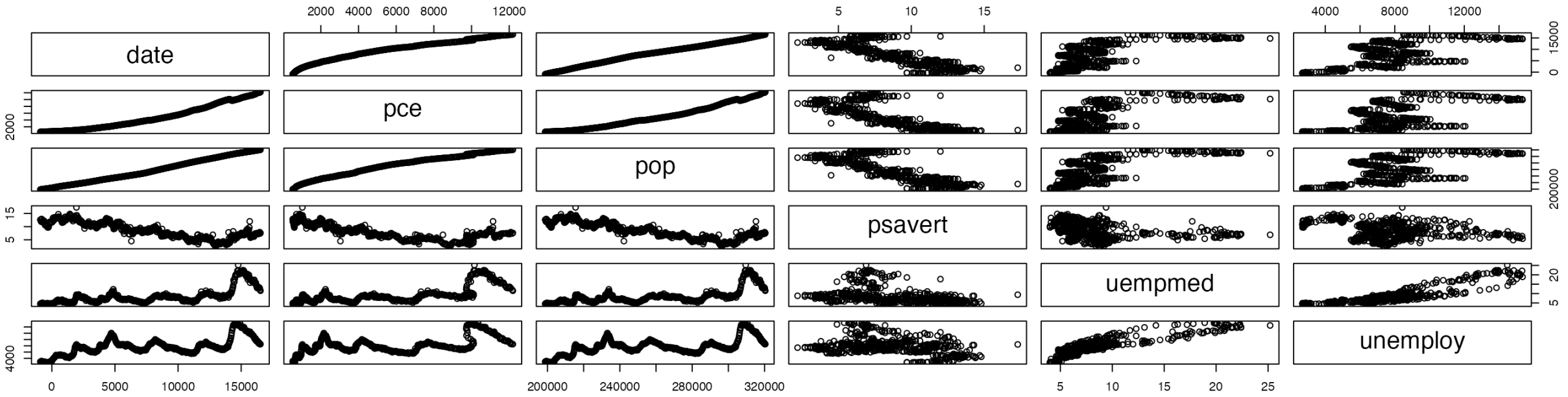
- L'application du fichier « intro_05 » vous permet de faire 3 actions de façon interactive:
  - *choisir les données à utiliser parmi de 3 data frames de ggplot2*
  - *imprimer un résumé statistique avec la fonction summary()*
  - *visualiser le contenu des dataframes avec la fonction plot()*

  Cette app suit également les bonnes pratiques et utilise des expressions réactives pour éviter la redondance du code. Cependant il y a trois bugs dans le code fourni. Pouvez-vous les trouver et réparer le code ? Le résultat de l'app doit ressembler à la page suivante.

Dataset

economics

```
      date                pce              pop          psavert         uempmed
 Min.   :1967-07-01   Min.   :  506.7   Min.   :198712   Min.   : 2.200   Min.   : 4.000
 1st Qu.:1979-06-08   1st Qu.: 1578.3   1st Qu.:224896   1st Qu.: 6.400   1st Qu.: 6.000
 Median :1991-05-16   Median : 3936.8   Median :253060   Median : 8.400   Median : 7.500
 Mean   :1991-05-17   Mean   : 4820.1   Mean   :257160   Mean   : 8.567   Mean   : 8.609
 3rd Qu.:2003-04-23   3rd Qu.: 7626.3   3rd Qu.:290291   3rd Qu.:11.100   3rd Qu.: 9.100
 Max.   :2015-04-01   Max.   :12193.8   Max.   :320402   Max.   :17.300   Max.   :25.200
    unemploy
 Min.   : 2685
 1st Qu.: 6284
 Median : 7494
 Mean   : 7771
 3rd Qu.: 8686
 Max.   :15352
```



Exercice 6

LE RENDU

- Créer une application shiny qui combine dans une même application les apps de l'exercice 1 avec celles de l'exercise 4 et 5. Mettez ce code sur github avec une capture d'écran du résultat obtenu en format pgn.
- À rendre, l'url de la répo github de ce projet