Guide complet de Shiny :

De débutant à expert

Créez des applications web interactives avec R

Auteur : Viny Presty NAKAVOUA

Date de création : 04/12/2024

SOMMAIRE

1 Introduction à R Shiny

R Shiny est un package puissant qui permet de créer des applications web interactives directement à partir de R. Développé par RStudio en 2012, Shiny facilite la construction d'interfaces utilisateur dynamiques sans nécessiter de connaissances approfondies en HTML, CSS ou JavaScript.

1.1 Qu'est-ce que Shiny ?

Shiny est un framework d'application web qui permet aux utilisateurs de R de transformer leurs analyses et visualisations en applications interactives accessibles via un navigateur web. Il se compose de deux parties principales :

1. **UI (User Interface)** : La partie visible de l'application avec laquelle l'utilisateur interagit.
2. **Server** : La partie backend où se déroulent les calculs et le traitement des données.

1.2 Avantages et cas d'utilisation

Les applications Shiny offrent de nombreux avantages pour la visualisation et l'analyse de données :

- **Accessibilité** : Les utilisateurs peuvent explorer les données via une interface web intuitive, sans nécessiter de connaissances en R.
- **Interactivité en temps réel** : Les visualisations et analyses se mettent à jour dynamiquement selon les interactions de l'utilisateur.
- **Personnalisation** : Les utilisateurs peuvent ajuster les paramètres pour obtenir des insights spécifiques.
- **Automatisation** : Les calculs complexes s'effectuent en arrière-plan, délivrant des résultats instantanés.
- **Partage facilité** : Les applications peuvent être facilement partagées, favorisant la collaboration.

1.3 Prérequis et installation

Pour commencer avec Shiny, vous aurez besoin de :

1. R installé sur votre ordinateur
2. RStudio (IDE recommandé)

Vous pouvez télécharger R depuis https://cran.r-project.org/

Pour une aide à l'installation, vous pouvez consulter cette vidéo YouTube : https://www.youtube.com/watch?v=I-KZoK6Bc7E

2 Création d'une application shiny

2.1 Structure d'une application Shiny

Une application Shiny se compose généralement de deux fichiers principaux :

1. `ui.R` : Définit l'interface utilisateur
2. `server.R` : Contient la logique et les calculs

Alternativement, vous pouvez utiliser un seul fichier `app.R` qui combine les deux.

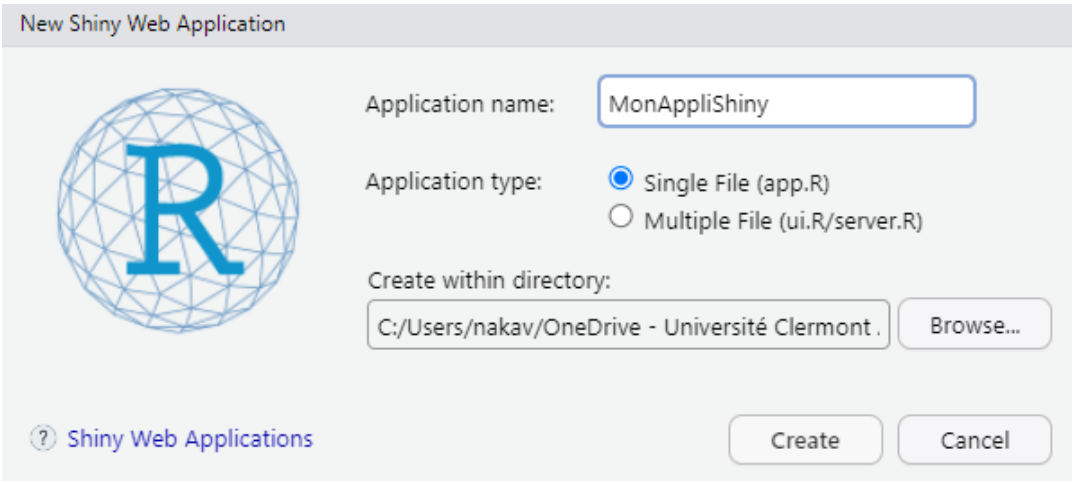
2.2 Création d'un projet Shiny

Pour créer une nouvelle application Shiny dans RStudio :

1. Allez dans File → New Project → New Directory → Shiny Web Application
2. Donnez un nom à votre projet (par exemple, "MonAppliShiny")
3. Choisissez le répertoire de sauvegarde
4. Cliquez sur "Create Project"

RStudio créera automatiquement les fichiers nécessaires pour commencer votre application Shiny.

Lors de la création d'un nouveau projet Shiny dans RStudio, vous avez deux options principales pour la structure de votre application :



2.2.1 Option 1 : Un seul fichier app.R

Dans cette configuration, tout le code de votre application (UI et server) est contenu dans un seul fichier `app.R`.

Pour exécuter l'application : 1. Ouvrez le fichier `app.R` 2. Cliquez sur le bouton "Run App" en haut à droite de l'éditeur RStudio

2.2.2 Option 2 : Deux fichiers séparés (ui.R et server.R)

Cette approche sépare l'interface utilisateur (UI) et la logique du serveur en deux fichiers distincts.

Pour exécuter l'application : 1. Assurez-vous que vous êtes dans le bon répertoire de travail contenant `ui.R` et `server.R` 2. Dans la console R, exécutez la commande :

```
shiny::runApp()
```

2.3 Construction de l'interface web avec le fichier ui.R

Le fichier `ui.R` (ou la partie UI dans `app.R`) définit l'interface utilisateur de l'application Shiny. Cette interface est visible par l'utilisateur et lui permet d'interagir avec l'application. Voici les points clés :

- L'interface est construite sans nécessiter de connaissances en HTML.
- La fonction `fluidPage()` est utilisée pour définir la structure de la page web.
- `mainPanel()` gère la partie droite de la page, où s'affichent généralement les résultats.

Exemple de structure de base :

```
ui <- fluidPage(
  titlePanel("Titre de l'application"),
  sidebarLayout(
    sidebarPanel(
      # Éléments d'entrée (inputs)
    ),
    mainPanel(
      # Éléments de sortie (outputs)
    )
  )
)
```

2.4 Écriture du programme dans le fichier server.R

Le fichier `server.R` (ou la partie server dans `app.R`) contient la logique de l'application et gère les calculs sous R. Caractéristiques principales :

- Il crée les sorties et met à jour les résultats de l'interface.
- Peut inclure des instructions exécutées à chaque lancement de l'application, comme :
  - Chargement de packages
  - Importation de jeux de données
  - Téléchargement de ressources (ex: cartes)
  - Calculs préliminaires

Structure de base du server :

```
server <- function(input, output) {
  # Logique de l'application
  # Création des outputs en fonction des inputs
}
```

La partie server réagit aux interactions de l'utilisateur avec l'interface, effectue les calculs nécessaires et met à jour les résultats affichés dans l'UI.

2.5 Mise à disposition de l'application Shiny

2.5.1 Test local

Avant publication, testez l'application sur votre ordinateur : - Utilisez les boutons "RunApp" ou "ReloadApp" dans RStudio.

2.5.2 Publication en ligne

Pour rendre l'application accessible à tous sans installation de R :

1. Cliquez sur "Publish Application" dans RStudio.
2. Créez un compte ShinyApps (première fois).
3. Choisissez une formule (gratuite pour jusqu'à 5 applications).
4. Générez un Token d'authentification sur ShinyApps.io.
5. Copiez et collez le Token dans RStudio.
6. Publiez l'application.

3 Les inputs et outputs

L'interface utilisateur (UI) se compose de deux types d'éléments principaux :

3.1 Inputs

- Widgets d'entrée (sliders, menus déroulants, boutons, cases à cocher, champs de texte, etc.)
- Placés généralement à gauche dans `sidebarPanel()`

3.2 Outputs

- Éléments de sortie (graphiques, tableaux, zones de texte, etc.)
- Généralement affichés à droite dans `mainPanel()`.

Les outputs sont définis dans le fichier `server.R` ou dans la partie server de `app.R`. Leur intégration dans l'interface utilisateur se fait en deux étapes :

3.2.1 Dans le fichier ui.R (ou partie UI)

Utilisez une fonction `*Output()` appropriée au type d'objet à afficher :

- `plotOutput()` pour les graphiques
- `tableOutput()` pour les tableaux
- `textOutput()` pour du texte

3.2.2 Dans le fichier server.R (ou partie server)

Définissez le contenu du composant en utilisant une fonction `render*()` correspondante. Cette fonction est associée à un élément de la liste `output`, dont le nom doit correspondre à l'identifiant utilisé dans l'UI.

Exemple :

```r
# Dans ui.R
ui <- fluidPage(
  plotOutput("monGraphique")
)

# Dans server.R
server <- function(input, output) {
  output$monGraphique <- renderPlot({
    # Code pour générer le graphique
  })
}
```

Cette structure assure que les outputs sont réactifs et se mettent à jour automatiquement en fonction des interactions de l'utilisateur avec les inputs.

Voici une correspondance entre les fonctions outputs et les fonctions render* à utiliser dans Shiny :

Fonction Output	Fonction Render
plotOutput	renderPlot
tableOutput	renderTable
textOutput	renderText
uiOutput	renderUI
imageOutput	renderImage
dataTableOutput	renderDataTable
printOutput	renderPrint

Chaque fonction render* produit un type spécifique de sortie qui correspond à la fonction *Output associée dans l'interface utilisateur. Par exemple, renderPlot() est utilisé pour générer des graphiques qui seront affichés via plotOutput() dans l'UI.

3.3 Fonctionnement

- Les interactions avec les inputs sont détectées et transmises au server.
- Le server traite les données et met à jour les outputs correspondants.
- Chaque input et output doit avoir un identifiant unique pour être utilisé dans la partie server.

4 Exemple d'application shiny

On va créer une application qui permet à l'utilisateur de visualiser interactivement la distribution des différentes caractéristiques des fleurs d'iris (longueur et largeur des sépales et pétales) sous forme d'histogrammes, en ajustant le nombre de bins(classes).

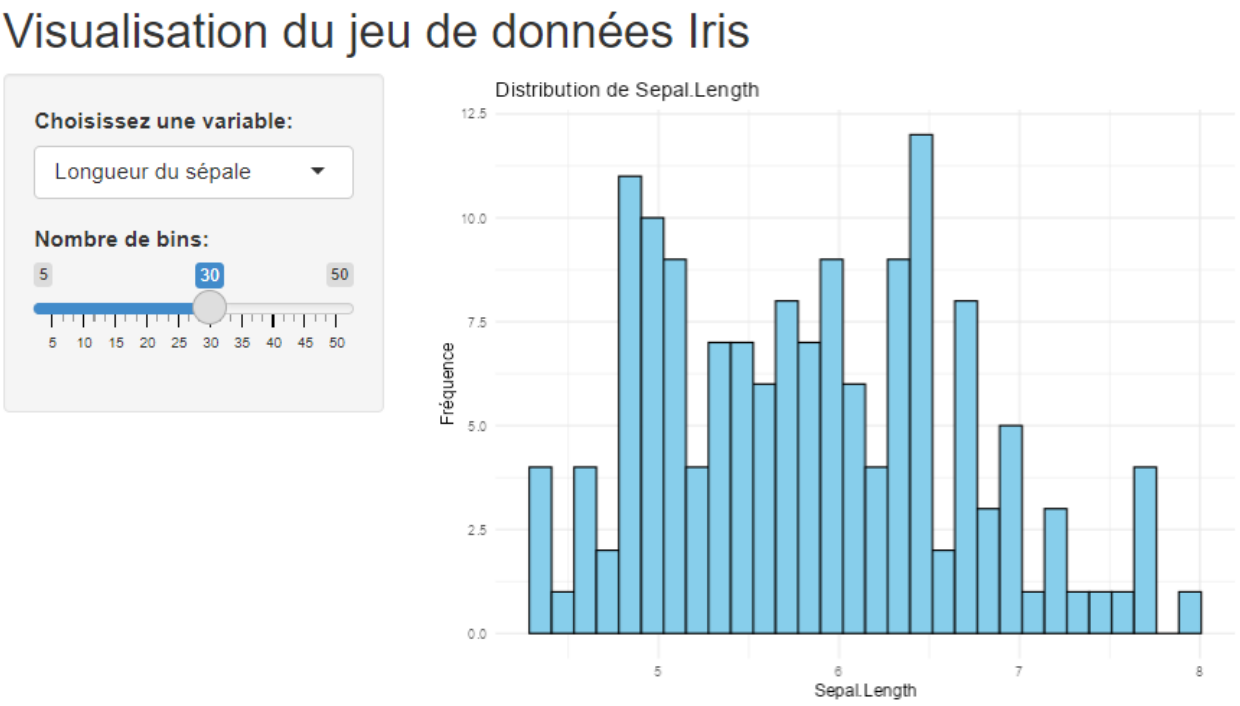


Figure 1 - Première application shiny.

4.1 Chargement des packages nécessaires

Pour cette application, j'ai premièrement besoin du package shiny, qui permet d'utiliser les fonctions permettant de créer l'application puis j'ai aussi besoin du package ggplot2 vue que dans mon application je vais utiliser un graphique qui l'histogramme.

```
library(shiny)
library(ggplot2)
```

4.2 Définition de l'interface utilisateur (UI)

```r
# Dans ui.R
ui <- fluidPage(
  # Titre de l'application
  titlePanel("Visualisation du jeu de données Iris"),

  # Mise en page avec barre latérale
  sidebarLayout(
    # Panneau latéral pour les contrôles
    sidebarPanel(
      # Menu déroulant pour choisir la variable
      selectInput("variable", "Choisissez une variable:",
                  choices = c("Longueur du sépale" = "Sepal.Length",
                              "Largeur du sépale" = "Sepal.Width",
                              "Longueur du pétale" = "Petal.Length",
                              "Largeur du pétale" = "Petal.Width"),
                  selected = "Sepal.Length"),

      # Curseur pour ajuster le nombre de bins de l'histogramme
      sliderInput("bins", "Nombre de bins:",
                  min = 5, max = 50, value = 30)
    ),

    # Panneau principal pour afficher le graphique
    mainPanel(
      # Sortie du graphique
      plotOutput("distPlot")
    )
  )
)
```

4.3 Définition du serveur

```
server <- function(input, output) {

 # Création du graphique réactif
 output$distPlot <- renderPlot({
  # Sélection de la variable en fonction de l'entrée de l'utilisateur
  x <- iris[[input$variable]]

  # Création de l'histogramme avec ggplot2
  ggplot(iris, aes_string(x = input$variable)) +
   geom_histogram(bins = input$bins, fill = "skyblue", color = "black") +
   labs(title = paste("Distribution de",  input$variable),
      x = input$variable,
      y = "Fréquence") +
   theme_minimal()
 })
}
```

4.4 Exécution de l'application Shiny

```
# exécution en local
shinyApp(ui = ui, server = server)
```

5 Application shiny complexe

Il est possible de construire des applications plus complexes. Nous considérons un second exemple qui permet de choisir un jeu de données, de sélectionner une variable quantitative puis, sur les données de cette variable, calcule sa médiane et construit un histogramme ainsi qu'un estimateur de la densité. Pour l'histogramme, on pourra faire varier le nombre de classes, la couleur et ajouter ou non une courbe de lissage en choisissant le mode de lissage. L'interface de cette application est disponible en figure 4.4 et l'application est disponible à l'adresse https://statavecr.shinyapps.io/MonAppliShiny2/.

L'objectif ici est de partir de l'exemple précédent qui prend en compte le jeu de données iris et l'histogramme, puis donc d'y ajouter l'autre jeu de données ainsi que les autres options, ainsi qu'en ajoutant des titres en utilisant des balises html.

A toi de jouer !!!

Vous trouverez le corrigé de cet exercice sur ce lien : https://github.com/VinyPrestyNakavoua/time_series/blob/main/RStatistiqueDataSciences.pdf, qui vous permettra d'accepter à mon compte github où vous pourrez télécharger le livre « R pour la statistique et la science des données », le corrigé se trouve à la page 135.

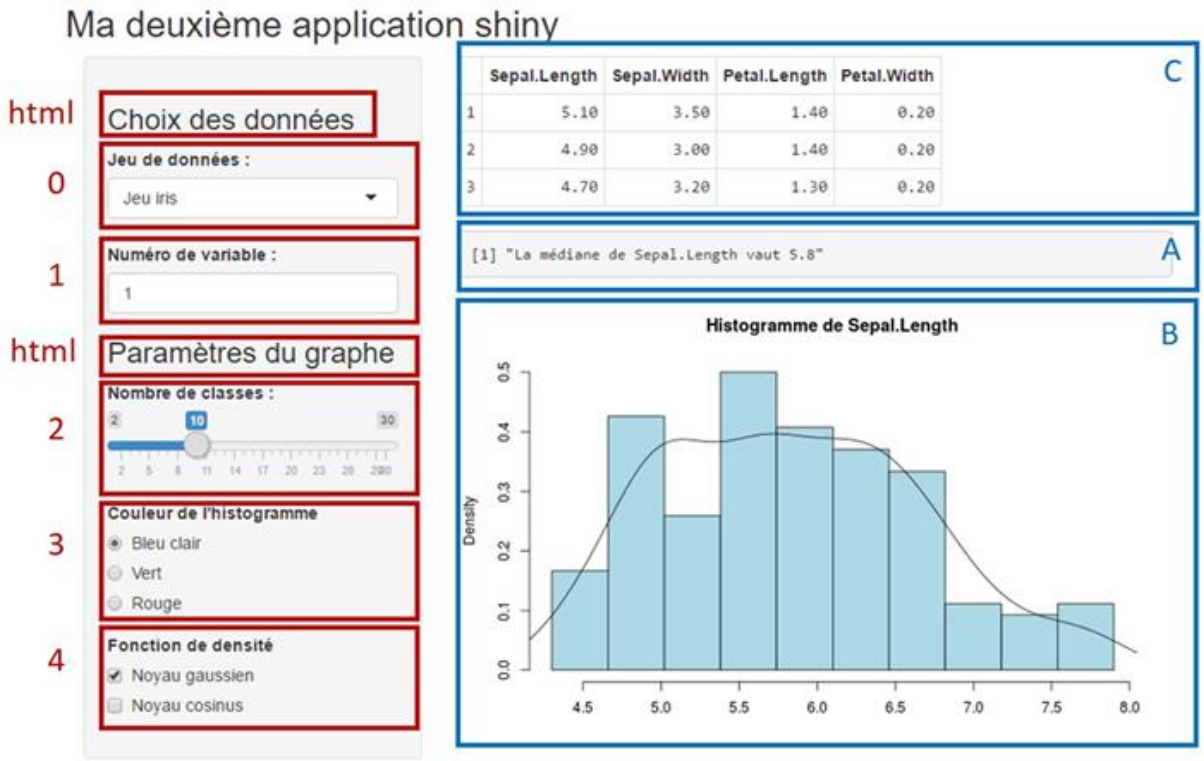


Figure 2 - Deuxième application shiny.

6 Ressources complémentaires pour Shiny

6.1 Documentation officielle

- Site officiel Shiny de Posit
- Documentation RDocumentation du package Shiny

6.2 Communauté et forums

- **Posit Community** : Forum principal pour obtenir de l'aide
- **Stack Overflow** : Tag "shiny" pour poser des questions techniques
- **Discord Shiny** : Serveur communautaire pour échanger avec d'autres développeurs
- **Groupe Google ShinyApps.io** : Support spécifique à la plateforme d'hébergement

6.3 Tutoriels et cours en ligne recommandés

- Tutoriel officiel Shiny de RStudio
- Chaîne YouTube "Formation R Shiny pour les débutants"
- Formations INRAE sur Shiny
- Tutoriels sur le site Posit

6.4 Cheatsheet

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

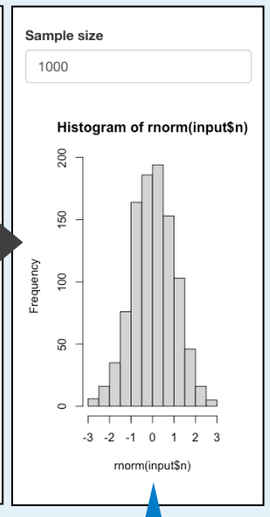
Refer to UI inputs with **input$<id>** and outputs with **output$<id>**

```
# 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.

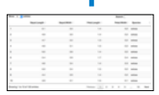
Share

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.

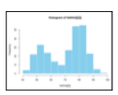


DT::**renderDataTable(**expr, options, searchDelay, callback, escape, env, quoted, outputArgs**)**

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**)**

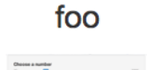
plotOutput(outputId, width, height, click, dblclick, hover, brush, inline**)**

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**)**



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.

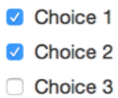
Inputs

Collect values from the user.

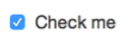
Access the current value of an input object with **input$<inputId>**. Input values are **reactive**.

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

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



checkboxGroupInput(inputId, label, choices, selected, inline, width, choiceNames, choiceValues**)**



checkboxInput(inputId, label, value, width**)**

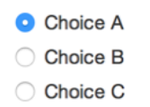
dateInput(inputId, label, value, min, max, format, startview, weekstart, language, width, autoclose, datesdisabled, daysofweekdisabled**)**

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

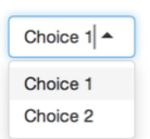
fileInput(inputId, label, multiple, accept, width, buttonLabel, placeholder**)**

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

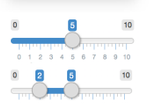
passwordInput(inputId, label, value, width, placeholder**)**



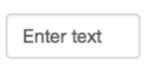
radioButtons(inputId, label, choices, selected, inline, width, choiceNames, choiceValues**)**



selectInput(inputId, label, choices, selected, multiple, selectize, width, size**)** Also **selectizeInput()**



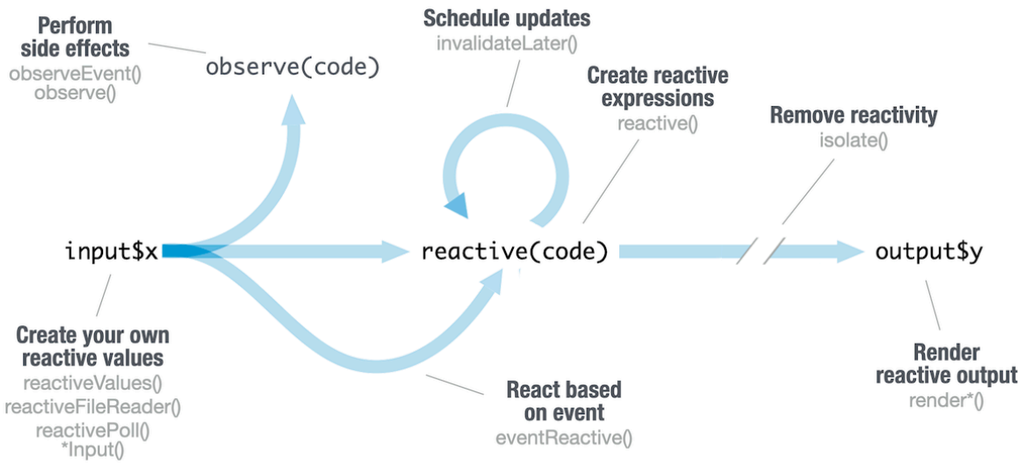
sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post, timeFormat, timezone, dragRange**)**



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")
)
```

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

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

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","")
)
```

Returns HTML

```
## <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>
```

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>"))
```

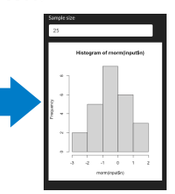
IMAGES

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

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",
  ...
 )
)
```

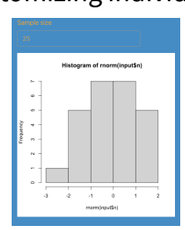


bootswatch_themes() Get a list of themes.

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.

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()

One Two Three
First tab content.

navset_pill()

One Two Three
First tab content.

navset_underline()

One Two Three
First tab content.

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()**