République du Sénégal

Un Peuple- Un But –Une Foie

Ministère de l'Economie du Plan et de la Coopération

****

ANSD

Agence national de la Statistique et de la Démographique

****

ensae

Ecole nationale de la Statistique et de l'Analyse économique Pierre NDIAYE

THÈME

Tableau de bord avec R shiny

Réalisé par :

Tamsir NDONG

Jean-Luc BATABATI

Ange Rayan RAHERINASOLO

Elèves Ingénieur statisticien économiste

Enseignant :

M. Aboubacar HEMA

Sommaire

Avant-propos

Créée en 2008, l'École Nationale de la Statistique et de l'Analyse Économique (ENSAE) est une grande école de statistique à caractère sous-régional située à Dakar (capitale du Sénégal). Elle constitue une direction de l'Agence Nationale de la Statistique et de la Démographie (ANSD), qui est la structure principale du Système Statistique Nationale du Sénégal.
Les élèves Ingénieurs Statisticiens Economistes qui suivent une formation de ISE (03 ans pour le cycle court et 05 ans pour le cycle long) à l'ENSAE se doivent de suivre le module Projet statistique sous R pour apprendre à utiliser et à traiter les données statistiques avec ce logiciel et à synthétiser les données à travers des graphiques afin de faciliter leur comprehension.

INTRODUCTION

À l'ère du numérique, la visualisation interactive des données est devenue un levier incontournable pour la prise de décision. Que ce soit dans les entreprises, les institutions publiques ou les milieux académiques, la capacité à transformer des données brutes en informations exploitables est un atout stratégique majeur. C'est dans ce contexte que R Shiny s'impose comme une solution puissante et accessible pour concevoir des tableaux de bord dynamiques et interactifs.

R Shiny, développé par la communauté R, permet de créer facilement des applications web sans pour autant recourir à des langages comme HTML, CSS ou JavaScript. Grâce à son interface intuitive, il offre aux analystes et data scientists la possibilité de déployer rapidement des outils de visualisation sur mesure, tout en exploitant pleinement l'écosystème R pour l'analyse de données.

Ce document a pour objectif de présenter les fondamentaux de la création de tableaux de bord avec R Shiny, d'en illustrer les avantages à travers des exemples concrets, et de fournir les bonnes pratiques pour construire des interfaces efficaces et esthétiques.

1 Chargement des packages nécessaires

Avant toute chose, nous veillerons à installer tous les packages qui servirons dans la suite puis les charger.

```
packages <- c("tidyverse","shiny","bslib","shinydashboard", "datasets",
              "DT", "rsconnect")

for (package in packages) {
  if (!requireNamespace(package, quietly = TRUE)) {
    install.packages(package)
  }
  library(package, character.only = TRUE)
}
```

- Le package **tidyverse** permet de **manipuler**, **analyser** et **visualiser** des données de manière cohérente grâce à une collection de packages tels que dplyr, ggplot2, tidyr, entre autres.
- Le package **shiny** permet de **créer des applications web interactives** pour explorer, analyser et visualiser des données en temps réel.
- Le package **bslib** permet de **personnaliser facilement l'apparence** d'une application Shiny à l'aide de thèmes Bootstrap modernes.
- Le package **shinydashboard** est une extension de Shiny qui **simplifie la création de tableaux de bord interactifs et attrayants**.
- Le package **datasets** donne accès à **divers jeux de données standards** (par exemple : iris, mtcars, airquality, etc.), utiles pour les démonstrations et les tests.
- Le package **DT** permet de **transformer des dataframes en tableaux dynamiques et interactifs**, avec des fonctionnalités comme le tri, la recherche ou la pagination.
- Le package **rsconnect** permet de **déployer des applications Shiny et des documents R Markdown** sur des serveurs web, notamment sur la plateforme shinyapps.io.

2 Généralités sur R Shiny

Pour créer une application avec R Shiny, il faut au préalable installer le package shiny.

2.1 Principes d'une application Shiny

Une application **Shiny** est composée de deux parties essentielles :
l'interface utilisateur (UI) et **la logique du serveur**.

- L'**UI** définit la **mise en page** ainsi que les **éléments interactifs** de l'application (boutons, graphiques, menus, etc.).
- Le **serveur** gère la **logique métier**, **traite les données** et **génère les sorties** à afficher à l'utilisateur.
- L'interaction entre l'UI et le serveur permet de créer des **applications réactives**, c'est-à-dire capables de répondre automatiquement aux actions de l'utilisateur.

Pour une meilleure organisation du code, il est recommandé de structurer l'application Shiny autour d'au moins trois fichiers :

- **ui.R** : contient le code de l'interface utilisateur.

- `server.R` : contient la logique de traitement côté serveur.
- `app.R` : combine les deux composants précédents et **lance l'application**.

2.2 Création d'un fichier R Shiny

Pour créer un fichier **R Shiny**, il suffit de suivre les étapes suivantes :

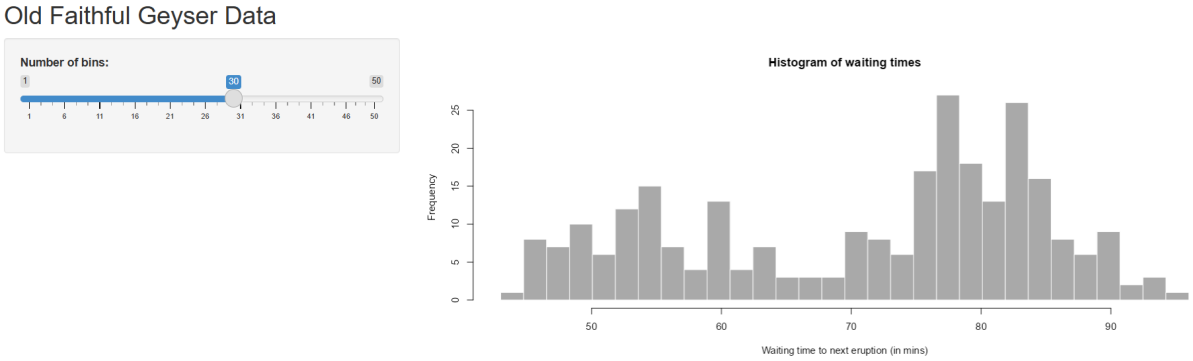
- Dans la **barre de menu**, cliquer sur **File**.
- Puis, sélectionner **New File**, puis **Shiny Web App**.
- Une boîte de dialogue s'affiche :
  - Saisir le **nom de l'application**.
  - Choisir le **type d'application** à créer : un fichier unique `app.R` ou deux fichiers séparés `ui.R` et `server.R`.
  - Préciser le **répertoire de destination** à l'aide du champ *Browse*.
  - Cliquer sur **Create** pour valider.

Une fois l'application créée, un **fichier app.R** s'ouvre automatiquement dans l'éditeur. Ce fichier contient une structure de base que nous pouvons adapter à notre projet.

```
#
# This is a Shiny web application. You can run the application by clicking
# the 'Run App' button above.
#
# Find out more about building applications with Shiny here:
#
#    https://shiny.posit.co/
#

library(shiny)

# Define UI for application that draws a histogram
ui <- fluidPage(

    # Application title
    titlePanel("Old Faithful Geyser Data"),

    # Sidebar with a slider input for number of bins
    sidebarLayout(
        sidebarPanel(
            sliderInput("bins",
                        "Number of bins:",
                        min = 1,
                        max = 50,
                        value = 30)
        ),

        # Show a plot of the generated distribution
        mainPanel(
           plotOutput("distPlot")
        )
```

```
    )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

    output$distPlot <- renderPlot({
        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = 'darkgray', border = 'white',
             xlab = 'Waiting time to next eruption (in mins)',
             main = 'Histogram of waiting times')
    })
}

# Run the application
shinyApp(ui = ui, server = server)
```

En compilant ce code nous avons le résultat ci dessous.



2.3 Parties principales

Une application Shiny a besoin d'au moins les 4 lignes suivantes :

```
library(shiny) # Pour le chargement du package Shiny

ui <- fluidPage() #pour la création de l'interface utilisateur

server <- function(input, output){} #pour la partie serveur

shinyApp(ui, server)  # Lancement de l'application
```

```
#crée l'application avec l'ui et la logique de serveur précédemment créées.
```

Notons que la fonction **`fluidPage()`** permet de créer une page **Shiny fluide**, qui s'adapte automatiquement à la taille de la fenêtre du navigateur. Il existe également d'autres types de pages, comme **`fixedPage()`**, qui, contrairement à **`fluidPage()`**, possède une taille fixe et ne s'ajuste pas à la taille de la fenêtre.

3 Structure d'une application Shiny

3.1 Création de l'interface utilisateur :

3.1.1 Les Layouts(dispositions)

Un **layout** dans R Shiny désigne la **structure visuelle** de l'application, c'est-à-dire la manière dont les éléments de l'interface utilisateur sont organisés à l'écran.

L'interface est généralement construite à partir de **fonctions imbriquées**, selon la logique suivante :

1. Une fonction définissant la **disposition générale**, comme `fluidPage()` (la plus courante).

2. Des **panneaux de mise en page** tels que `sidebarPanel()`, `mainPanel()` ou `tabPanel()` pour organiser le contenu.

3. Pour créer des onglets, on utilise les fonctions `tabsetPanel()` (disposition horizontale) ou `navlistPanel()` (disposition verticale), afin de diviser l'affichage en plusieurs sections indépendantes.

```r
library(shiny)


ui <- fluidPage(
  titlePanel("Notre application Shiny"),

  sidebarLayout(
    sidebarPanel("Panneau latéral"),
    mainPanel(
      tabsetPanel(
        tabPanel("Onglet 1", "Contenu de l'onglet 1"),
        tabPanel("Onglet 2", "Contenu de l'onglet 2")
      )
    )
  )
)
# Run the application
shinyApp(ui = ui, server = server)
```

Voici comment se présentent la disposition horizontale :

3.1.2 Autres dispositions possibles

Shiny offre plusieurs fonctions pour créer des interfaces variées :

- `navbarPage()` : permet d'ajouter une **barre de navigation horizontale** en haut de l'application.
- `navbarMenu()` : permet de **regrouper plusieurs `tabPanel()`** sous un même menu déroulant.
- `column()` : permet de **disposer des éléments en colonnes**, utile pour créer des mises en page personnalisées.

À noter : Les packages `bslib` et `shinydashboard` offrent encore plus de flexibilité pour créer des interfaces modernes et structurées, grâce à leurs propres fonctions de mise en page.

3.2 Inputs-Outputs

3.2.1 Inputs (widgets)

Les **inputs** permettent aux utilisateurs d'interagir avec l'application. Ils sont généralement placés dans le `sidebarPanel()`.

Parmi les plus courants :

- `textInput()` : permet à l'utilisateur de saisir un texte.
- `numericInput()` : permet de saisir une valeur numérique.

Toutes les fonctions d'input partagent au minimum deux arguments :

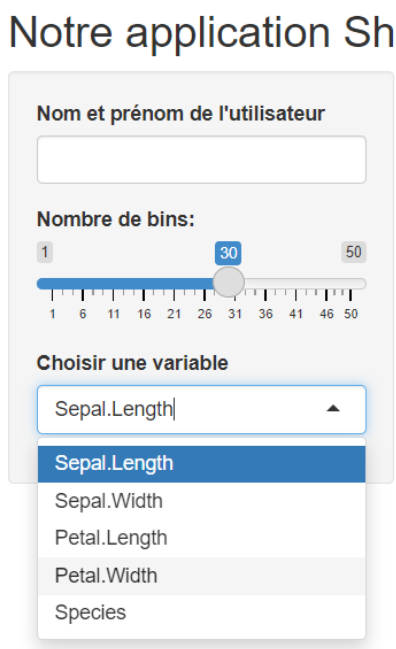
- `inputId` : identifiant unique de l'élément, utilisé pour récupérer sa valeur dans le serveur.
- `label` : texte descriptif affiché à l'utilisateur.

Dans l'exemple suivant :

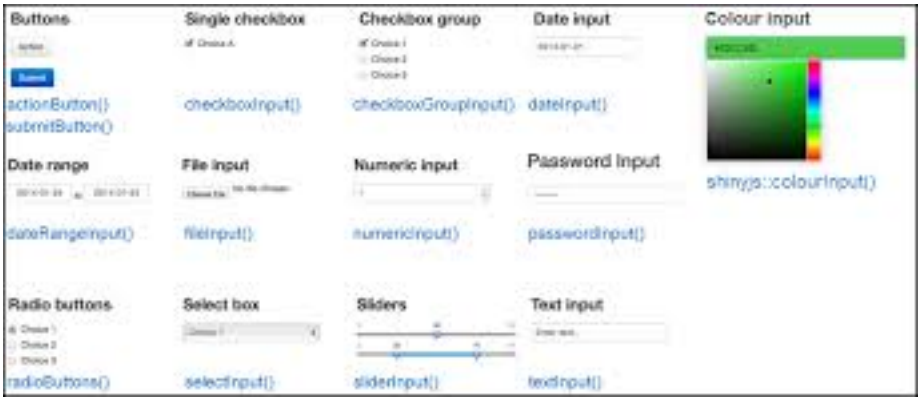
- `textInput()` est utilisé pour demander le **nom et prénom** de l'utilisateur,
- `selectInput()` pour **choisir les variables** à afficher depuis la base **Iris**,
- `sliderInput()` pour **ajuster le nombre de barres** dans un histogramme que nous allons définir.

```
 library(shiny)
data(iris)

# Interface utilisateur
ui <- fluidPage(
  titlePanel("Notre application Shiny"),
  sidebarLayout(
    sidebarPanel(
      textInput(inputId = 'text', label = "Nom et prénom de l'utilisateur"),
      sliderInput("curseur", "Nombre de bins:",
                  min = 1, max = 50, value = 30),
      selectInput(inputId = 'var', label = 'Choisir une variable', choices = names(iris))
    ),
    mainPanel(
      # Affichage du texte de l'utilisateur
      textOutput("text_output"),
      # Affichage du graphique basé sur la variable sélectionnée
      plotOutput("hist_plot")
```

```
    )
  )
)

# Partie serveur
server <- function(input, output) {}

# Lancer l'application
shinyApp(ui = ui, server = server)
```



Il existe une varieté d'inputs pour des actions spécifiques.



3.2.2 Outputs

Les **outputs** permettent de créer des **espaces réservés** pour afficher les résultats générés par l'application dans le **panneau principal**.

Ces sorties peuvent être de **n'importe quel type d'objet R** : graphiques, tableaux, textes, résumés statistiques, etc.

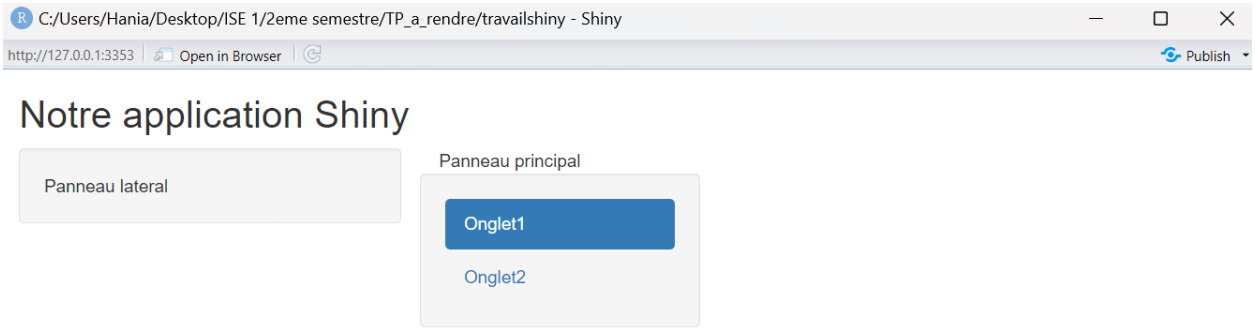
Shiny propose différentes fonctions de sortie, chacune adaptée à un type spécifique :

- `textOutput()` : pour afficher **du texte dynamique**.
- `verbatimTextOutput()` : pour afficher **du texte brut** (résumé, message, structure...).
- `plotOutput()` : pour afficher **des graphiques**.
- `DTOutput()` : pour afficher **des tableaux interactifs** avec le package `DT`.

Dans notre exemple, chaque sortie est placée dans un **onglet distinct** pour une organisation claire et lisible.

```
library(DT)
interface <- fluidPage(
  sidebarLayout(
  sidebarPanel(),
mainPanel (
 navlistPanel(
    tabPanel("Nom et Prénom",textOutput(outputId = "name")),
    tabPanel("Visualisation des données",DTOutput(outputId = "iris_data")),
    tabPanel("Statistiques",verbatimTextOutput(outputId = 'summary')),
    tabPanel("Histogramme",plotOutput(outputId = 'hist'))

      )
    )
  )
)
# Run the application
shinyApp(ui = interface, server = server)
```

Voici comment s'affichent nos différents onglets correspondants aux outputs cités plus haut:



Voici quelques outputs et leurs fonctions:

```r
library(knitr)
kable(
  data.frame(
    "Type d'Output" = c("Texte", "Texte HTML", "Tableau", "Tableau interactif", "Graphique de base", "G
    "Fonction UI" = c("textOutput()", "htmlOutput()", "tableOutput()", "DT::dataTableOutput()", "plotOu
    "Fonction server" = c("renderText()", "renderUI()", "renderTable()", "DT::renderDataTable()", "rend
    "Description" = c(
      "Affiche du texte brut dynamique",
      "Affiche du texte HTML formaté",
      "Affiche une table statique",
      "Tableau avec tri, recherche, pagination",
      "Graphique statique (base R ou ggplot2)",
      "Graphique interactif (zoom, survol)",
      "Affiche du texte brut (résultats de code)",
      "Interface générée dynamiquement",
      "Affiche une image",
      "Téléchargement de fichier"
    )
  ),
  format = "markdown",
  align = "l"
)
```

Type.d.Output	Fonction.UI	Fonction.server	Description
Texte	textOutput()	renderText()	Affiche du texte brut dynamique
Texte HTML	htmlOutput()	renderUI()	Affiche du texte HTML formaté
Tableau	tableOutput()	renderTable()	Affiche une table statique
Tableau interactif	DT::dataTableOutput()	DT::renderDataTable()	Tableau avec tri, recherche, pagination
Graphique de base	plotOutput()	renderPlot()	Graphique statique (base R ou ggplot2)
Graphique interactif	plotlyOutput()	renderPlotly()	Graphique interactif (zoom, survol)
Texte/verbatim	verbatimTextOutput()	renderPrint()	Affiche du texte brut (résultats de code)
UI personnalisée	uiOutput()	renderUI()	Interface générée dynamiquement
Image	imageOutput()	renderImage()	Affiche une image
Téléchargement	downloadButton()	downloadHandler()	Téléchargement de fichier

3.2.3 Quelques précisions sur les outputs

Tout comme les fonctions d'input, les fonctions d'**output** possèdent un argument essentiel : `outputId`, qui doit être **unique** pour chaque sortie.

Ce `outputId` joue un rôle crucial : il permet au **serveur** d'associer un contenu (graphique, texte, tableau...) à l'espace de sortie défini dans l'interface. C'est pourquoi son **unicité** est indispensable.

> **Important :** Jusqu'à présent, aucune sortie ne s'affiche automatiquement dans l'application. C'est uniquement le **serveur** qui déclenche leur génération et leur affichage en fonction des instructions définies dans sa fonction.

3.3 Le serveur

Le **serveur** représente le **cœur logique** de l'application Shiny. C'est lui qui traite les **interactions** de l'utilisateur issues de l'interface (inputs) et qui génère les **sorties dynamiques** (outputs) en réponse.

Pour cela, Shiny met à disposition plusieurs fonctions essentielles :

- `renderXXX()` : permet de **lier une sortie** (`outputId`) à une expression R (ex. : `renderPlot()`, `renderText()`...).
- `reactive()` : crée des **objets réactifs**, qui se mettent automatiquement à jour lorsqu'un input change.
- `observe()` : permet de **surveiller des changements** d'inputs et d'exécuter du code en conséquence (sans produire de sortie visible).

3.3.1 Les fonctions `renderXXX()`

Dans le **serveur**, les fonctions `renderXXX()` servent à **générer dynamiquement les contenus** affichés dans les zones d'output définies dans l'interface utilisateur.

Chaque fonction `renderXXX()` correspond à une fonction `XXXOutput()` utilisée dans l'UI. Par exemple :

- `renderPlot()` est lié à `plotOutput()` pour afficher un **graphique**,
- `renderText()` est lié à `textOutput()` pour afficher **du texte**,
- `renderDT()` est lié à `DTOutput()` pour afficher une **table interactive**.

Voici différentes fonctions `render()` et leurs fonctions :

Fonction `render()` côté `server`	Fonction `output` côté `ui`	Utilisation principale
`renderText()`	`textOutput("id")`	Afficher du texte dynamique (valeurs, résultats simples)
`renderPrint()`	`verbatimTextOutput("id")`	Afficher du texte brut (résultat de code, résumés, etc.)
`renderTable()`	`tableOutput("id")`	Afficher un tableau statique
`renderPlot()`	`plotOutput("id")`	Afficher un graphique statique (R base, ggplot2, etc.)
`renderUI()`	`uiOutput("id")` ou `htmlOutput("id")`	Générer dynamiquement une interface (textes, boutons, etc.)
`renderImage()`	`imageOutput("id")`	Afficher une image (locale ou générée)
`renderDataTable()` *(package DT)*	`dataTableOutput("id")`	Afficher un tableau interactif (tri, recherche, etc.)
`renderPlotly()` *(package plotly)*	`plotlyOutput("id")`	Afficher un graphique interactif (zoom, survol, etc.)

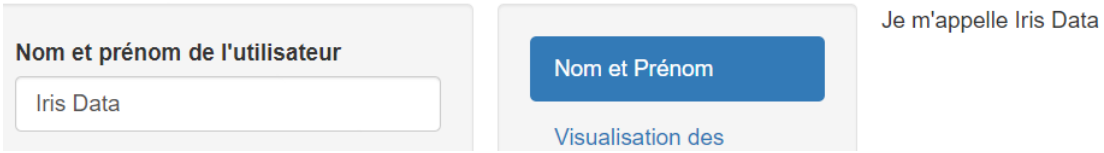
Ces fonctions exécutent du **code R** chaque fois qu'un input change, assurant une mise à jour réactive des sorties.

Dans l'exemple qui suit :
- `renderText()` permet d'afficher le **nom et prénom** saisis,
- `renderDT()` affiche les **données filtrées d'Iris**,
- `renderPlot()` génère un **histogramme interactif**.

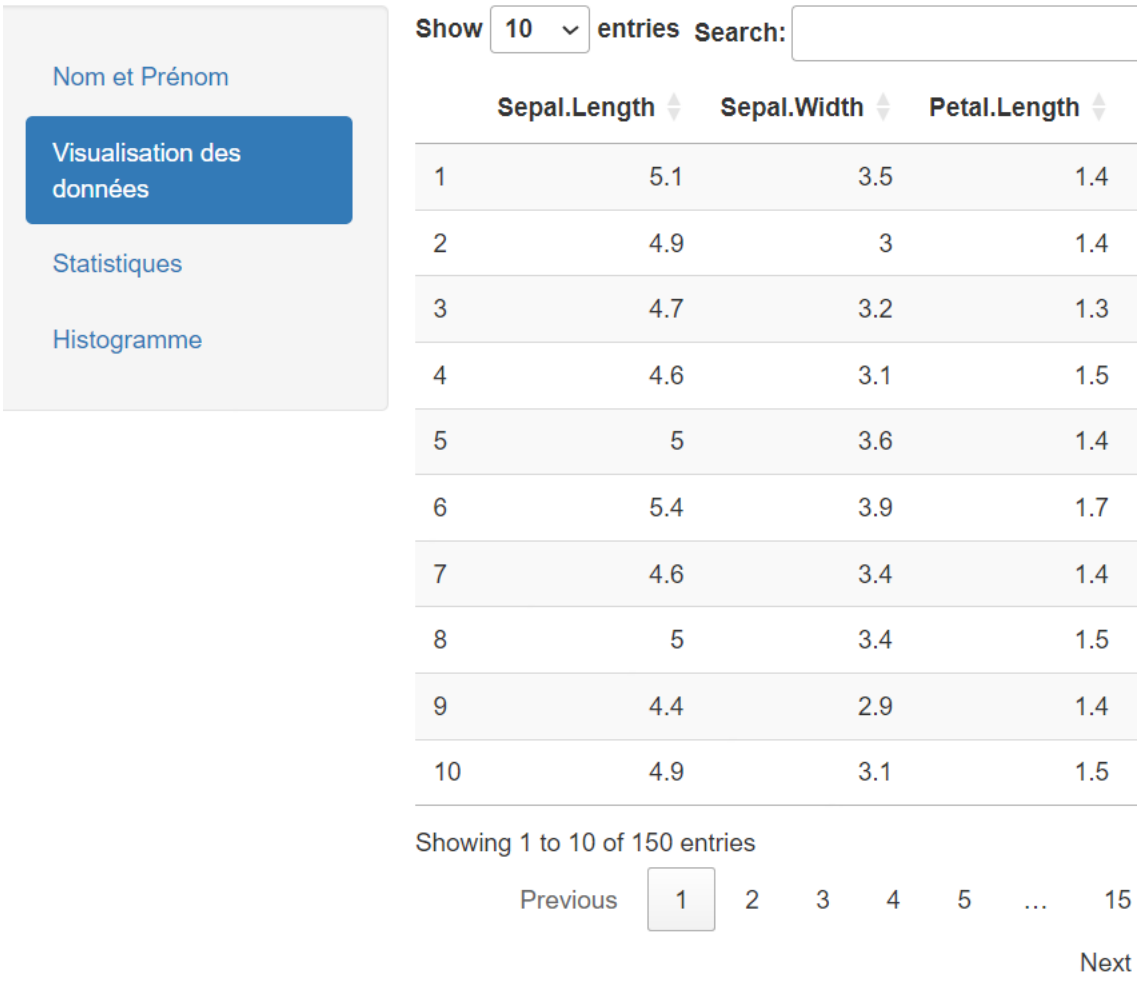
```
server <- function(input, output) {
  #texte
  output$name=renderText(paste0("Je m'appelle " ,input$text))
  output$iris_data=renderDT({iris})
  #resumé statistiques
  output$summary=renderPrint({
    summary(iris) })
```

```
  #Histogrammes
  output$hist=renderPlot({ hist(iris[[input$var]],
breaks = seq(min(iris[[input$var]]), max(iris[[input$var]]), length.out = input$slider),
main = paste("Histogramme de", input$var),
xlab = input$var,
ylab = "Fréquence") })
}
```

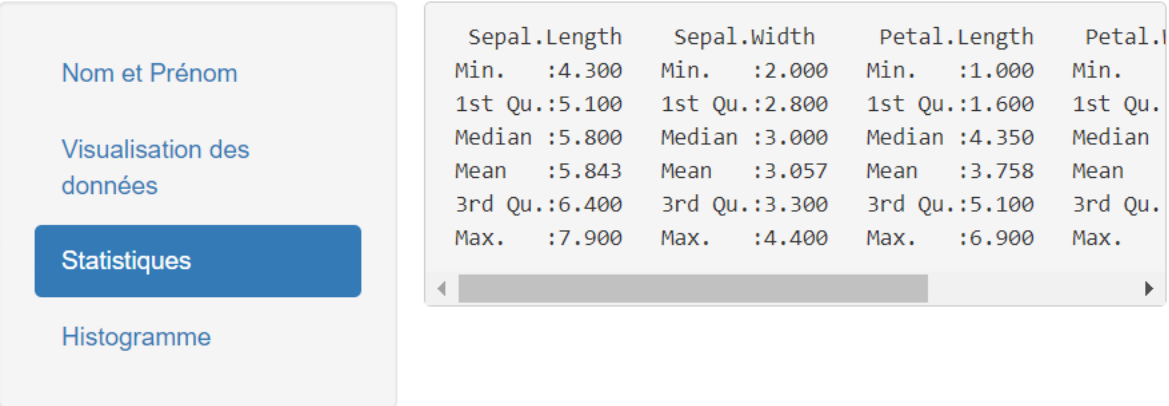
Sortie Texte :



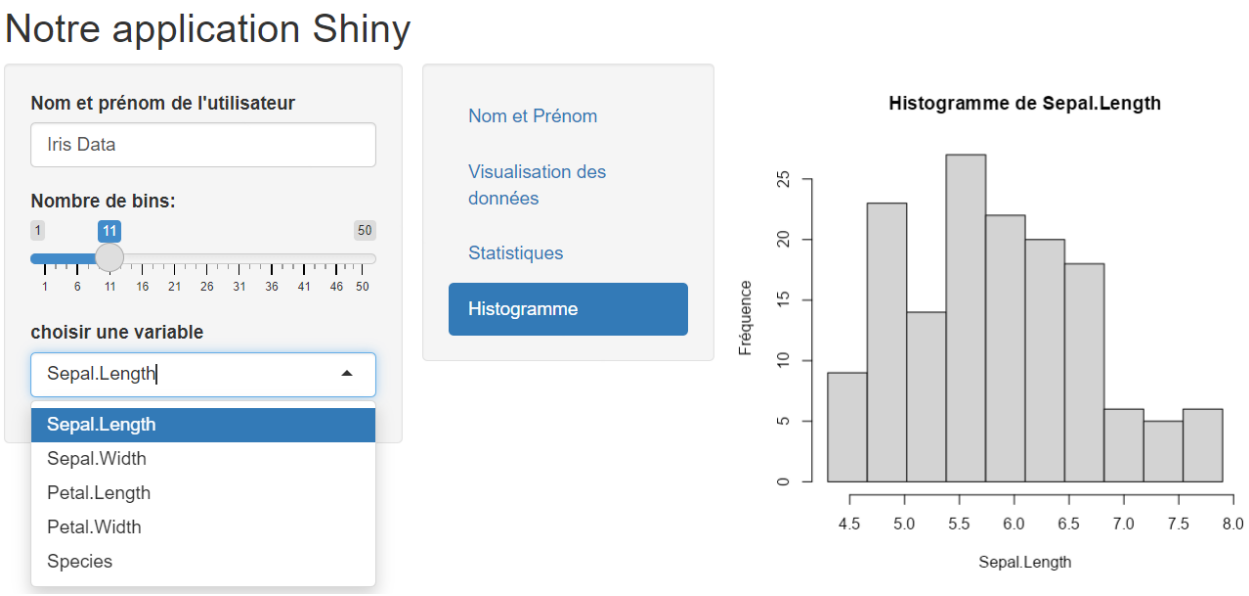
Sortie Table :



Sortie Statistiques :



Sortie Histogramme :



3.4 La fonction `reactive({})`

La **réactivité** est au cœur du fonctionnement des applications Shiny. Elle permet à l'interface de se **mettre à jour automatiquement** dès qu'un élément (input, donnée...) change.

Dans ce contexte, la fonction `reactive()` (ou `reactiveValues()`) est utilisée côté serveur pour créer des **objets réactifs**. Ces objets :

- Sont recalculés **automatiquement** lorsque leurs dépendances changent,
- Évitent les recalculs inutiles et facilitent la gestion des données dynamiques.

  Les objets créés avec `reactive()` sont souvent utilisés à l'intérieur des fonctions `renderXXX()` pour produire des sorties actualisées en temps réel.

```
library(DT)
server <- function(input, output, session) {
    df=reactive({
    iris
    })#Création de la variable reactive
```

```
  output$iris_data=renderDT({
    df() #Utilisation de la variable dans une fonction render
  })
}
```

3.4.1 Autres fonctions réactives

En plus de `reactive()`, Shiny propose d'autres fonctions pour gérer les interactions et la logique réactive :

- `observeEvent()` : déclenche une action **lorsqu'un événement spécifique** (comme un clic sur un bouton) se produit.
- `observe()` : **surveille des expressions réactives**, mais **ne produit pas de sortie** visible. Utile pour exécuter des opérations comme des enregistrements ou envois d'alertes.
- `isolate()` : permet d'**empêcher un recalcul automatique** en **brisant temporairement la réactivité**. Cela évite des mises à jour inutiles dans certains cas.

4 Amélioration de l'apparence d'une application Shiny

Améliorer l'apparence d'une application Shiny ne repose pas uniquement sur la **structure** (panneaux, onglets, etc.), mais aussi sur le **style visuel**, notamment à travers les **thèmes**.

4.1 Le package `shinythemes`

Le moyen le plus simple de personnaliser le style d'une application est d'utiliser le package **`shinythemes`**, qui propose une **large sélection de thèmes prédéfinis**.

Pour appliquer un thème, il suffit d'ajouter la fonction `shinytheme("nom_du_thème")` dans l'UI.

Exemple : ici, nous utilisons le thème `"journal"` pour un rendu

```r
library(shiny)
library(shinythemes)

ui <- fluidPage(
  theme = shinytheme("journal"),  # Change le thème ici (ex : "cosmo", "darkly", "flatly")
  titlePanel("Exemple de thème Shiny"),
  sidebarLayout(
    sidebarPanel(
      helpText("Ceci est un panneau latéral.")
    ),
    mainPanel(
      h3("Contenu principal"),
      textOutput("texte")
    )
  )
)

server <- function(input, output, session) {
  output$texte <- renderText({
    "Bienvenue dans l'application avec thème !"
  })
}

shinyApp(ui, server)
```

La liste de tous les thèmes disponibles se trouve sur le site library(Bootswatch).

4.2 Thème personnalisé avec un fichier CSS

Pour utiliser un **thème personnalisé** qui ne fait pas partie du package `shinythemes`, il est possible d'ajouter un **fichier CSS** à votre application.

1. Créer un sous-dossier nommé **`www`** dans le répertoire de votre application Shiny.

2. Placer votre fichier **CSS** (par exemple `style.css`) dans ce dossier.

3. Ajouter la ligne suivante dans la fonction `fluidPage()` de l'UI pour lier le fichier CSS :

```
ui <- fluidPage(theme = "theme.css",

)
```

NB : Il est **essentiel** de placer le fichier CSS dans un sous-dossier nommé **`www`**. C'est **la cause la plus fréquente d'erreur** si le thème personnalisé ne s'affiche pas correctement.

Shiny offre **de nombreuses possibilités** pour améliorer l'interface et l'expérience utilisateur.
En plus des thèmes via `shinythemes` ou CSS, il existe des **packages dédiés à la création d'interfaces avancées**, comme :

- **`shinydashboard`** : pour construire des tableaux de bord modernes avec des boîtes, des menus latéraux, des entêtes personnalisables, etc.

5 Shinydashboard

Le package **shinydashboard** est une extension du package **Shiny** qui permet de **concevoir des tableaux de bord interactifs et visuellement attrayants** au sein d'applications web développées avec R. Il fournit un ensemble de composants et d'outils facilitant la création d'interfaces utilisateur structurées, avec une disposition typique de tableau de bord (menus latéraux, en-têtes, boîtes, onglets, etc.), adaptée à des analyses complexes et interactives.

5.1 Structure d'un dashboard

Tout comme une application développée avec **Shiny**, une application construite avec **Shinydashboard** se compose de trois parties principales : l'**interface utilisateur** (`ui`), la **partie serveur** (`server`) et la **commande d'exécution** (`shinyApp()`).

La logique reste identique à celle d'une application Shiny classique :

- La **partie UI** définit la mise en page générale, les éléments visuels (menus, boîtes, graphiques, etc.) ainsi que les interactions avec l'utilisateur.
- La **partie serveur** contient la logique de traitement, permettant de générer les résultats à afficher de manière dynamique.

Le package **shinydashboard** introduit cependant des fonctions spécifiques comme `dashboardPage()`, `dashboardHeader()`, `dashboardSidebar()` et `dashboardBody()` pour structurer l'interface selon un format de tableau de bord.

5.2 Les principales dispositions

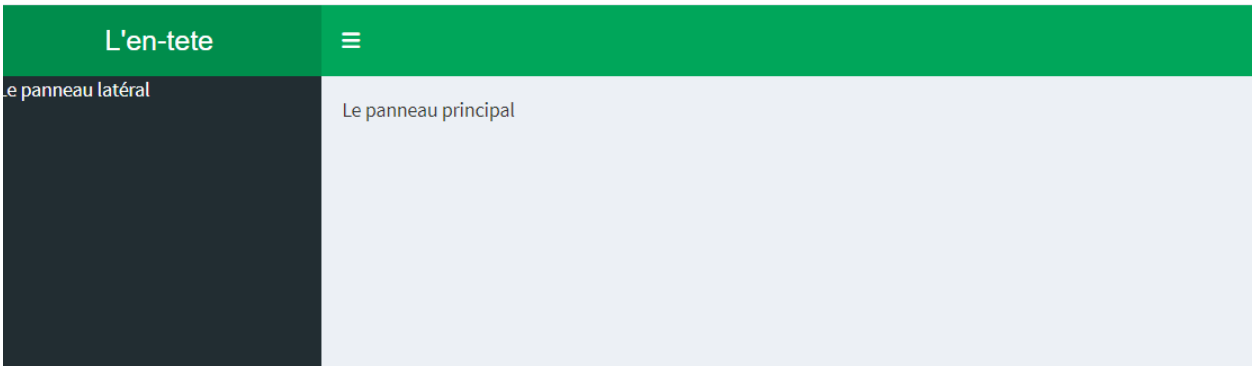
Dans la partie **interface utilisateur**, la mise en page d'un tableau de bord avec **shinydashboard** est gérée à l'aide de la fonction `dashboardPage()`, qui organise l'interface en trois zones principales :

- `dashboardHeader()` : définit l'en-tête de l'application (barre supérieure) ;
- `dashboardSidebar()` : crée le panneau latéral, souvent utilisé pour la navigation ou les filtres ;
- `dashboardBody()` : constitue la zone principale où s'affichent les contenus et résultats (graphiques, tableaux, etc.).

En plus de ces trois éléments essentiels, `dashboardPage()` accepte des arguments optionnels comme :

- `title` : pour afficher un titre dans l'onglet du navigateur ;
- `skin` : pour personnaliser le thème de couleur principal de l'application (par défaut : **bleu**, mais aussi disponible en **noir**, **rouge**, **jaune**, **vert**, **violet**).

```
ui <- dashboardPage(
  dashboardHeader(title="L'en-tete"),
  dashboardSidebar(title="Le panneau latéral "),
  dashboardBody("Le panneau principal "),
  skin="green"
)
server <- function(input, output) {
}
shinyApp(ui, server)
```



5.3 Dispositions supplémentaires

5.3.1 Barre latérale

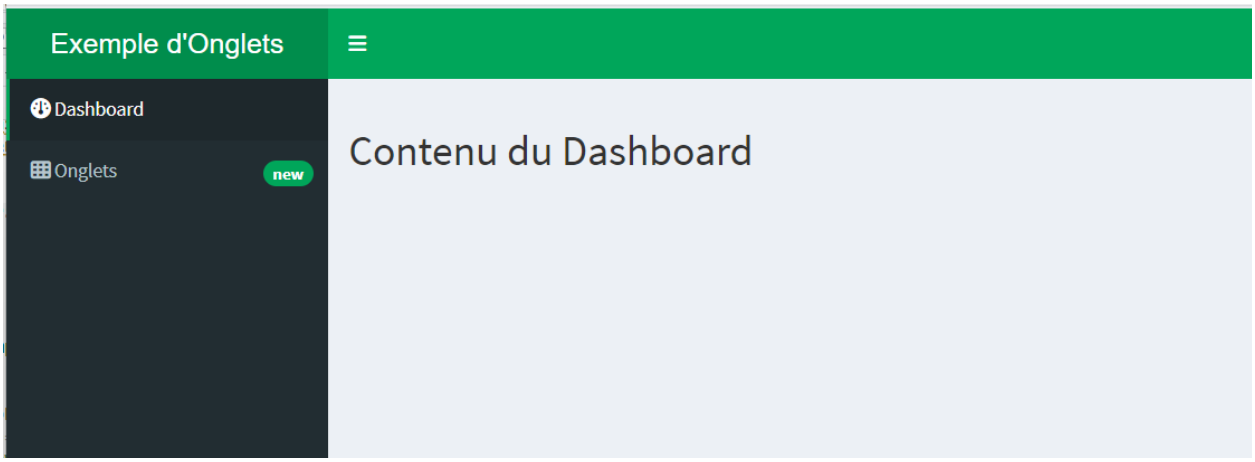
La **barre latérale** (`dashboardSidebar()`) contient les éléments permettant à l'utilisateur d'interagir avec la partie principale de l'application ou de naviguer entre différentes sections.

On y retrouve à la fois :

- des **éléments d'entrée classiques** (`sliderInput()`, `selectInput()`, etc.) ;
- des **menus de navigation**, sous forme d'onglets, créés avec la fonction `menuItem()`.

Chaque `menuItem()` déclaré dans le `dashboardSidebar()` correspond à un `tabItem()` défini dans le `dashboardBody()`. La liaison entre ces deux éléments se fait via l'argument `tabName`, qui doit être **identique et unique** de part et d'autre.

```
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  skin="green",
  dashboardHeader(title = "Exemple d'Onglets"),
  dashboardSidebar(
    sidebarMenu(
      menuItem("Dashboard", tabName = "dashboard", icon = icon("dashboard")),
      menuItem("Onglets", icon = icon("th"), tabName = "widgets",
               badgeLabel = "new", badgeColor = "green")
    )
  ),
  dashboardBody(
    tabItems(
      tabItem(tabName = "dashboard",
        h2("Contenu du Dashboard")
      ),
      tabItem(tabName = "widgets",
        h2("Contenu des Onglets")
      )
    )
  )
)

server <- function(input, output, session) {}

shinyApp(ui, server)
```



5.3.2 Corps ou partie principale :

Mis à part les `tabItem`,la partie principale peut contenir n'importe quel contenu standard de shiny mais, il est en plus possible de déclarer des box pouvant inclure différents types de contenus, ce qui est plus esthétique pour les tableaux de bord.

Cela se fait avec la fonction `box()`dans laquelle on peut integrer des éléments de type input ou output.

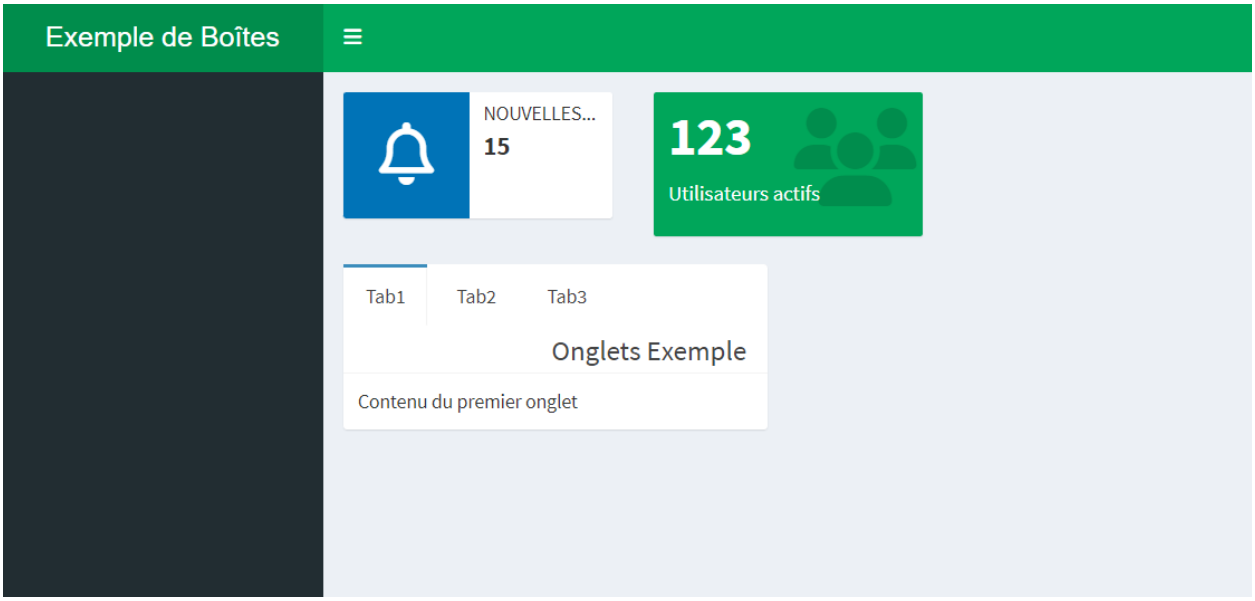
Il en existe trois autres types : `infoBox()`,`valueBox`,`tabBox()`.

La disposition des boites se fait avec `fluidrow()`ou `column()`.

L'exemple suivant montre la création de differents types de box avec affichages d'icônes issues de la bibliothèque **Font Awesome**, et intégration des onglets pour le type tabBox.

```
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  skin="green",
  dashboardHeader(title = "Exemple de Boîtes"),
  dashboardSidebar(),
  dashboardBody(
    fluidRow(
      # InfoBox
      infoBox("Nouvelles Notifications", 15, icon = icon("bell"), color = "blue"),

      # ValueBox
      valueBox(123, "Utilisateurs actifs", icon = icon("users"), color = "green"),

      # Box avec onglets (TabBox)
      tabBox(
        title = "Onglets Exemple",
        tabPanel("Tab1", "Contenu du premier onglet"),
        tabPanel("Tab2", "Contenu du deuxième onglet"),
        tabPanel("Tab3", "Contenu du troisième onglet")
      )
    )
  )
)
```

```r
server <- function(input, output, session) {}

shinyApp(ui, server)
```



5.3.3 Le server

Dans une application **ShinyDashboard**, le **serveur** conserve toutes les fonctions classiques de Shiny, telles que `render()`, `output()`, `reactive()`, `observe()`, etc. Ces fonctions sont cruciales pour définir le comportement interactif de l'application, traiter les données, générer des sorties et maintenir la réactivité de l'interface utilisateur.

Dans la partie **UI** de l'application, les éléments de contenu, tels que les onglets et les boîtes du tableau de bord, sont définis avec des fonctions comme `tabItem()` ou `box()`. C'est ici que l'on spécifie les sorties que l'on souhaite afficher (graphiques, tableaux, valeurs, etc.) en utilisant des fonctions de sortie telles que **`plotOutput()`, `tableOutput()`, `verbatimTextOutput()`**, et d'autres.

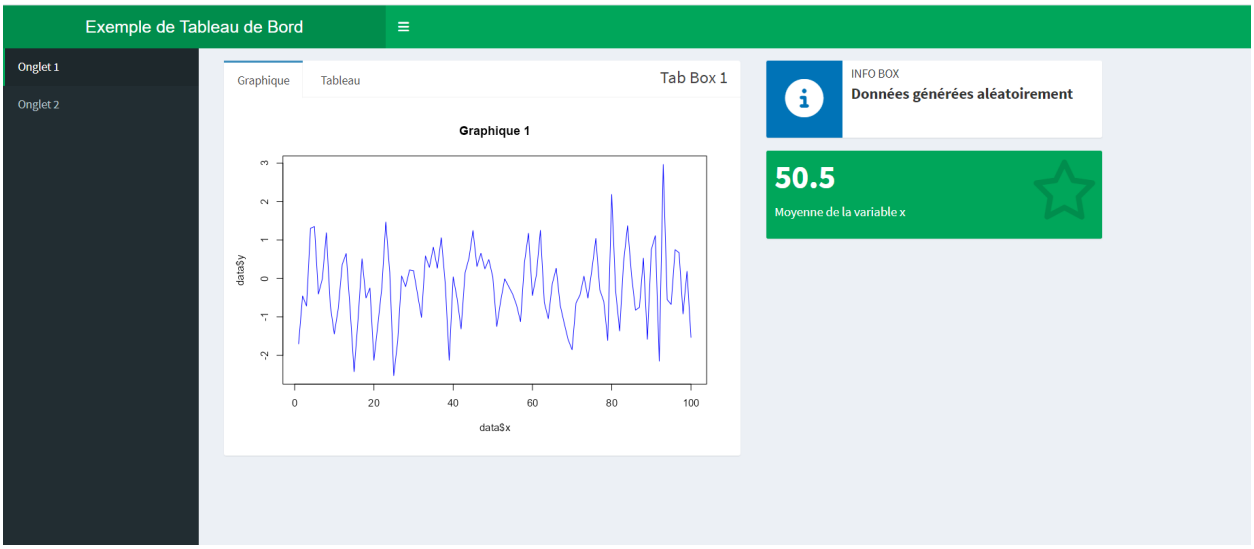
Côté **serveur**, des fonctions de rendu comme `renderPlot()`, `renderTable()`, `renderText()`, etc., sont utilisées pour générer les sorties spécifiées dans la partie UI.

En ce qui concerne la **réactivité**, chaque fois qu'il y a un changement dans les entrées réactives utilisées pour générer les sorties, le serveur est automatiquement déclenché pour recalculer et mettre à jour les résultats associés. Cela se fait notamment grâce à l'utilisation de **`observe()`** ou **`reactive()`**. Ce mécanisme garantit que l'interface utilisateur reste dynamique et que les contenus des **`tabItem`** ou **`box`** sont continuellement mis à jour en fonction des actions de l'utilisateur ou des modifications des données.

5.4 Exemple d'une application simple avec shinydashboard utilisant les éléments vus précédemment

```r
data <- data.frame(
  x = 1:100,
  y = rnorm(100)
)

ui <- dashboardPage(
  skin="green",
```

```
  dashboardHeader(title = "Exemple de Tableau de Bord", titleWidth = 450),
  dashboardSidebar(
    sidebarMenu(
      menuItem("Onglet 1", tabName = "tab1"),
      menuItem("Onglet 2", tabName = "tab2")
    )
  ),
  dashboardBody(
    tabItems(
      tabItem(tabName = "tab1",
              tabBox(
                title = "Tab Box 1",
                tabPanel("Graphique", plotOutput("plot1")),
                tabPanel("Tableau", tableOutput("table1"))
              ),
              infoBox(
                "Info Box",
                "Données générées aléatoirement",
                icon = icon("info-circle"),
                color = "blue"
              ),
              valueBox(
                mean(data$x),
                "Moyenne de la variable x",
                icon = icon("star"),
                color = "green"
              )
      ),
      tabItem(tabName = "tab2",
              tabBox(
                title = "Tab Box 2",
                tabPanel("Graphique", plotOutput("plot2"))
              )
      )
    )
  )
)

server <- function(input, output) {
  output$plot1 <- renderPlot({
    plot(data$x, data$y, main = "Graphique 1", type = "l", col = "blue")
  })
  output$table1 <- renderTable({
    head(data)
  })
  output$plot2 <- renderPlot({
    plot(data$y, data$x, main = "Graphique 2", pch = 19, col = "red")
  })
}

shinyApp(ui, server)
```


Exemple de Tableau de Bord
Onglet 1
Onglet 2
Graphique
Tableau
Tab Box 1
Graphique 1
data$y
data$x
INFO BOX
Données générées aléatoirement
50.5
Moyenne de la variable x

6 Publier une application Shiny

Lorsque l'on utilise la commande `runApp()`, R lance un petit serveur Shiny local sur notre ordinateur, ce qui nous permet de **tester l'application en local**. Cependant, cette méthode n'est pas adaptée si l'on souhaite **partager l'application publiquement** et la rendre accessible en ligne pour d'autres utilisateurs.

6.1 Hébergement sur shinyapp

Il existe plusieurs façons de publier une application Shiny, et l'une des plus simples consiste à utiliser le service **shinyapps.io**. Ce service permet d'héberger gratuitement jusqu'à **5 applications par utilisateur**.

Pour publier une application sur shinyapps.io, voici les étapes à suivre :

1. Créez un compte sur le site shinyapps.io et connectez-vous.
2. Une fois connecté, vous accédez à un **tableau de bord** où seront listées vos applications. La première étape consiste à installer le package **`rsconnect`** dans R, qui vous permettra de déployer votre application directement depuis votre environnement R.

```r
install.packages("rsconnect")
```

3. Ensuite, copier les lignes présentées dans la pop-up du type (ceci est un exemple avec le compte *Jean-R-Shiny-2025*) :

```r
#rsconnect::setAccountInfo(name='Jean-R-Shiny-2025',
                     #      token='6280C86118286E541729884 1A85EE625',
                     #      secret='ymUisnP7jRsyShTvGW0iAiUwnOGv98HnyMbcMd19')
```

4. Exécuter ces lignes sous R : vous êtes alors connecté à votre compte shinyapps.io depuis R.
5. Il suffit maintenant d'exécuter la ligne de code suivante (mise en commentaire).

```r
#rsconnect::deployApp("chemin_vers_le_dossier_de_votre_application")
```

6.2 Mise en ligne de l'application

Une fois votre application prête, elle peut être mise en ligne. Pour cela, vous devez placer dans votre dossier de travail **soit un fichier `app.R`**, soit **deux fichiers `ui.R` et `server.R`**.

Remarques importantes :

- Si vous avez déjà un compte sur **shinyapps.io**, vous pouvez lancer la publication en cliquant sur ***Publish Application...***, situé à droite du bouton ***Run App*** dans RStudio.
- RStudio vous demandera alors un **token** pour authentifier votre compte shinyapps.io. Il vous suffit de copier-coller ce token depuis votre compte shinyapps.io pour établir la connexion.
- La publication peut prendre quelques minutes, car shinyapps.io installe automatiquement tous les packages nécessaires au fonctionnement de votre application sur son serveur.
- Une fois l'opération terminée, votre application s'ouvrira automatiquement dans votre navigateur web.
- Si vous perdez l'URL de votre application, pas de panique : vous pouvez retrouver toutes vos applications dans votre tableau de bord sur shinyapps.io.

6.3 Mise à jour d'une application Shiny

Pour mettre à jour votre application, il suffit simplement de relancer la commande ***Publish Application...***. L'ancienne version sera alors automatiquement **remplacée par la nouvelle**.

CONCLUSION

6.4 Conclusion

R Shiny est un outil puissant et accessible pour créer des applications web interactives avec R. Il permet de concevoir des interfaces intuitives, de visualiser dynamiquement des données, et de partager des analyses en ligne.

Grâce à sa large communauté, ses nombreux packages complémentaires et ses possibilités d'extension, Shiny constitue une solution complète pour répondre aux besoins en visualisation, recherche ou communication de données.

Maîtriser Shiny représente donc un véritable atout pour tout professionnel souhaitant valoriser et diffuser ses analyses de manière interactive et professionnelle.

BIBLIOGRAPHIE et WEBOGRAPHIE