Découverte :

Rendre ses projets R plus accessibles grâce à Shiny

mer 24 Avr 2019 Thomas Denecker Découverte, Didacticiel 3

Bonjour à tous !

Vous avez un script que vous souhaitez partager avec une équipe expérimentale? Vous ne voulez pas que les utilisateurs modifient le code pour paramétrer votre programme? Vous codez avec R ? Alors cet article est fait pour vous ! Nous allons voir comment créer une application web avec R et permettre à votre utilisateur d'exécuter votre code sans le voir.

Shiny

Le package que nous utiliserons est shiny. Il est proposé par Rstudio (https://shiny.rstudio.com/) et disponible sur le CRAN. Ce package permet de construire des applications web très simplement sans connaissances particulières en HTML et CSS. Les fonctions que nous appellerons dans R vont être traduites en HTML. Par exemple, `h1('Un titre')` sera transformé en `<h1>Un titre</h1>`. Il n'est donc pas indispensable de savoir coder en HTML, mais des connaissances dans les langages web pourront vous être utiles dans des cas particuliers, puisqu'il est possible d'intégrer dans l'application shiny du code HTML brut.

Une application shiny se divise en 2 parties :

- l'UI : Il s'agit de l'interface utilisateur visible dans une page web. Nous pourrons y retrouver des graphes, des tableaux, du texte, etc. L'utilisateur pourra interagir avec cette interface par le biais de boutons, de sliders, de cases, etc.
- le serveur : Il s'agit de la « zone de travail ». Tous les calculs, préparations de données ou analyses que R réalisera seront faits côté serveur.

Nous allons voir dans cet article toutes les étapes pour créer une application complète. Elle sera capable de lire un fichier en fonction de paramètres enregistrés par l'utilisateur puis d'afficher :

- Un tableau avec de la coloration conditionnelle
- 4 graphiques obtenus par des approches différentes
  - Un classique réalisé avec R. Ce graphique sera paramétrable par l'utilisateur (couleur ou titre par exemple).
  - Un réalisé avec la librairie ggplot2
  - Un dynamique réalisé avec plotly
  - Un dynamique réalisé avec Google

L'ensemble du code permettant de réaliser l'application est disponible sur github : https://github.com/bioinfo-fr/bioinfo-fr_Shiny

Pré-requis

Toutes les étapes pour créer une application Shiny seront détaillées dans ce post. Connaître la syntaxe de R simplifiera grandement la lecture de l'article mais n'est pas indispensable.

Pour réaliser cette application, il vous faudra une version à jour de RStudio (plus simple que la console R). Pour l'installer, suivez les étapes suivantes (l'ordre est important) :

1. installer R : https://cran.r-project.org/
2. Installer RStudio : https://www.rstudio.com/products/rstudio/download/

Note : pour les utilisateurs de R les plus avancés, l'application peut être développée dans un environnement virtuel comme docker (sujet de mon prochain post).

Les données

Les données utilisées pour cette application proviennent du tableau IRIS regroupant des mesures sur des fleurs (disponible dans Rdataset et décrit ici https://archive.ics.uci.edu/ml/datasets/iris). Ce jeu de données est très utilisé pour illustrer les fonctions dans R et pour le *machine learning*. Le tableau est composé de 5 colonnes :

- la longueur des sépales ;
- la largeur des sépales ;
- la longueur des pétales ;
- la largeur des pétales ;
- l'espèce de fleurs.

Un fichier au format txt est disponible ici :
https://github.com/bioinfo-fr/bioinfo-fr_Shiny/blob/master/datasetIris.txt .

Les packages R

Les packages utilisés pour réaliser l'application sont disponibles sur le CRAN. Ils s'installent avec la commande : `install.packages()`.
Les packages que nous utiliserons sont :

- **shiny** [1] : Il permettra de construire l'application web
- **shinydashboard** [2]: Il permettra de créer une architecture dynamique à la page web avec une zone de titre, une menu rabattable et une zone principale
- **shinyWidgets** [3] : Il permettra de mettre un message d'alerte pour confirmer la lecture correcte du tableau
- **DT** [4] : Il permettra de créer un tableau dynamique avec de la coloration conditionnelle
- **plotly** [5] , **ggplot2** [6] et **googleVis** [7] : Ils nous permettront de réaliser des graphiques
- **colourpicker** [8] : Il permettra à l'utilisateur de sélectionner une couleur.

Nous utiliserons pour les installer et les charger un autre package : anylib [9]. Ce package est très pratique car il permet d'installer (si besoin) et de charger une liste de package. En plus, il a été créé par un des auteurs de Bioinfo-fr : Aurelien Chateigner. Que demander de plus!

```
install.packages("anyLib")
anyLib::anyLib(c("shiny", "shinydashboard", "shinyWidgets", "DT", "plotly", "ggplot2", "googleVis", "colourpicker"))
```

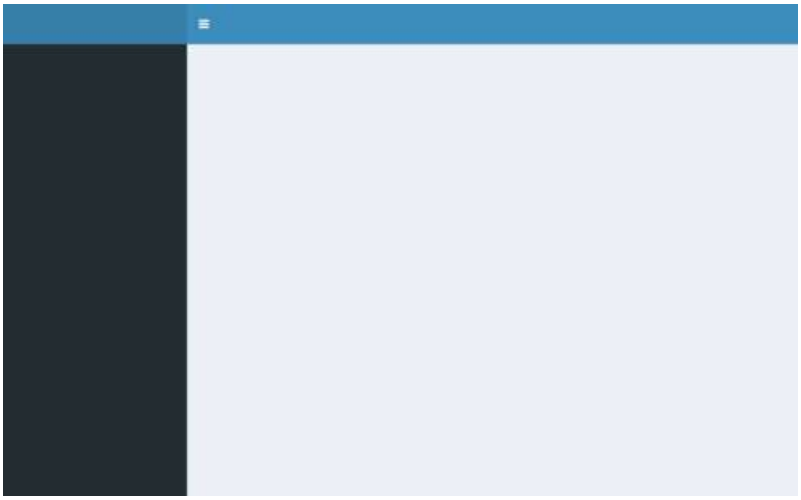
Création de l'architecture

Mise en place d'un dashboard

Pour mettre en forme notre application web (la partie UI visible par l'utilisateur), nous allons utiliser le package shinydashboard. La documentation est présente ici : https://rstudio.github.io/shinydashboard/index.html .

L'architecture minimale avec shinydasbord est zone de titre (bleue), une barre latérale (noir) et une zone principale (grise).

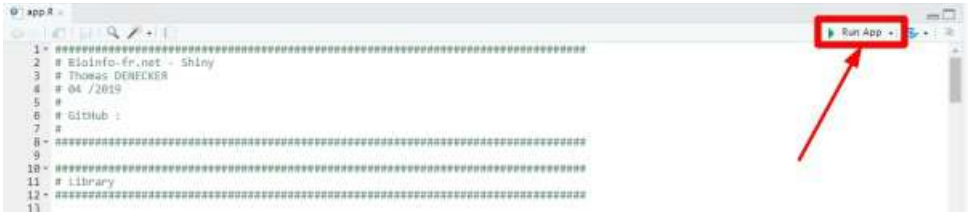
```
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody()
)

server <- function(input, output) { }

shinyApp(ui, server)
```



Visualisation de l'application

Test de l'application

Pour tester l'application, il faut sauvegarder le code puis appuyer sur le bouton `Run App` au dessus à droite de l'éditeur de texte de Rstudio. Un point important, Rstudio reconnaît par défaut les applications qui se nomme app.R. Je vous conseille vivement de nommer votre fichier app.R.



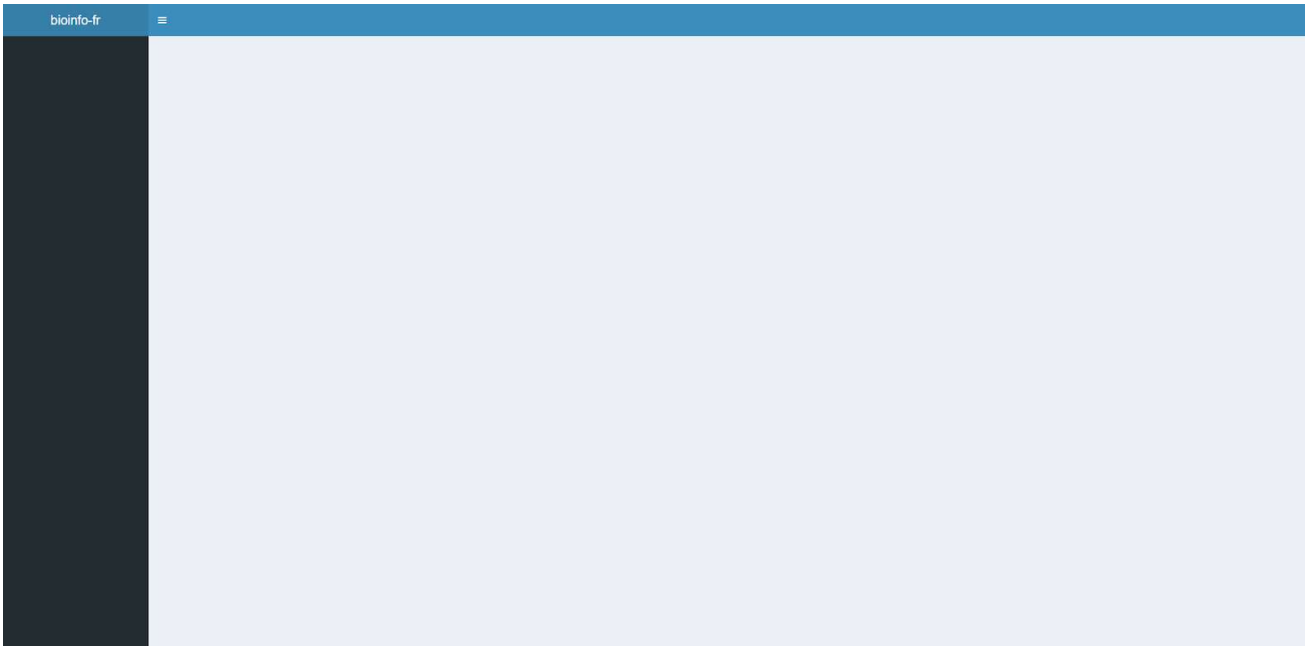
Si vous travaillez avec la console R, vous pouvez lancer la commande suivante :

```
runApp()
```

Ajouter un titre

Dans la fonction dashboardHeader, nous ajoutons un titre à l'application (ici bioinfo-fr). Ce titre sera affiché en haut à gauche.

```
ui <- dashboardPage(
  dashboardHeader(title = "bioinfo-fr"),
  dashboardSidebar( ),
  dashboardBody( )
)
```



Visualisation de l'application

Ajouter des pages

La première étape est d'ajouter des éléments (item) dans la barre de menu latérale (partie noire). Nous utilisons pour cela la fonction `dashboardSidebar`. Nous y ajoutons la fonction `sidebarMenu`qui contient les items du menu.

Ensuite, il faut indiquer que la partie body aura plusieurs pages (des `tabItems`). Chaque `tabItem` correspond à une page accessible par le menu. Le `menuItem` doit avoir le même nom que l'argument `tabName` de la fonction `tabItem` pour y accéder (exemple : readData). Dans chaque page, nous ajoutons un titre de niveau 1 (h1). Vous pouvez remarquer l'utilisation de la fonction `icon` (`icon = icon(...)`). L'argument de la fonction est un nom d'icône que nous pouvons trouver sur ces deux sites : https://fontawesome.com/ et https://getbootstrap.com/docs/4.3/components/alerts/ . En utilisant cette fonction, vous aurez une petite image (icônes) à gauche du nom de l'élément (par exemple un livre pour la lecture des données). Il est aussi possible de l'utiliser pour des boutons .

```
ui <- dashboardPage(
  dashboardHeader(title = "bioinfo-fr"),
  dashboardSidebar(
    sidebarMenu(
      menuItem("Lecture des données", tabName = "readData", icon = icon("readme")),
      menuItem("Visualisation des données", tabName = "visualization", icon = icon("poll"))
    )
  ),
  dashboardBody(
    tabItems(
      # Read data
      tabItem(tabName = "readData",
              h1("Lecture des données")
      ),

      # visualization
      tabItem(tabName = "visualization",
              h1("Visualisation des données")
      )
    )
  )
)
```

Visualisation de l'application

Création d'un lecteur de fichier

L'objectif est de proposer une interface simple pour lire un fichier dans l'application et qui permette à l'utilisateur de paramétrer la lecture et d'avoir une prévisualisation du fichier lu.

Importer un fichier

Pour importer un fichier, shiny propose la fonction `fileInput`. Il est possible de faire du "drag and drop" dans la zone de l'import ou de sélectionner un fichier dans l'explorateur de fichiers. Le type de fichier visible est paramétrable dans les arguments. Ici, nous utiliserons le paramétrage par défaut.

```
ui <- dashboardPage(
  dashboardHeader(title = "bioinfo-fr"),
  dashboardSidebar(
    sidebarMenu(
      menuItem("Lecture des données", tabName = "readData", icon = icon("readme")),
      menuItem("Visualisation des données", tabName = "visualization", icon = icon("poll"))
    )
  ),
  dashboardBody(
    tabItems(
      # Read data
      tabItem(tabName = "readData",
              h1("Lecture des données"),
              fileInput("dataFile",label = NULL,
                        buttonLabel = "Browse...",
                        placeholder = "No file selected")
      ),

      # visualization
      tabItem(tabName = "visualization",
              h1("Visualisation des données")
      )
    )
  )
)
```

Visualisation de l'application

Zone de paramétrage

Nous souhaitons maintenant paramétrer 3 points lors de la lecture du fichier : type de séparateur (virgule, tabulation, espace), type de quote (simple, double, aucune) et la présence/absence des noms de colonnes (header). Nous utilisons pour cela des radio boutons. La fonction utilisée est `radioButtons`. 5 arguments sont utilisés :

- **id** : identifiant du groupe de radio boutons (ici nous avons 3 groupes de radio boutons pour nos 3 paramètres),
- **label** : le titre présent au dessus du groupe de radio boutons,
- **choices** : les choix possibles dans le groupe de radio boutons. A noter, la zone située à gauche du "=" contient les informations qui seront affichées dans l'application alors que la partie droite indique ce que comprend R côté serveur. Pour le header par exemple, il sera affiché "Yes" côté UI et nous récupérerons côté serveur `TRUE` ("Yes" = TRUE) lorsque que nous récupérerons la valeur du radio bouton côté serveur.
- **Selected** : Nom du radio bouton sélectionné au lancement de l'application
- **inline = T** : pour avoir les radio boutons alignés

```
[...]
tabItem(tabName = "readData",
              h1("Lecture des données"),
              fileInput("dataFile",label = NULL,
                        buttonLabel = "Browse...",
                        placeholder = "No file selected"),

              h3("Parameters"),

              # Input: Checkbox if file has header
              radioButtons(id = "header",
                           label = "Header",
                           choices = c("Yes" = TRUE,
                                       "No" = FALSE),
                           selected = TRUE, inline=T),

              # Input: Select separator ----
              radioButtons(id = "sep",
                           label = "Separator",
                           choices = c(Comma = ",",
                                       Semicolon = ";",
                                       Tab = "\t"),
                           selected = "\t", inline=T),

              # Input: Select quotes ----
              radioButtons(id = "quote",
                           label= "Quote",
                           choices = c(None = "",
                                       "Double Quote" = '"',
                                       "Single Quote" = "'"),
                           selected = "", inline=T)

      ),

[...]
```

Visualisation de l'application

Zone de prévisualisation

Dans cette zone, nous allons visualiser les premières lignes du fichier que nous souhaitons lire. Il faut donc :

- Créer une zone d'affichage dans l'UI
- Lire les données côté serveur et envoyer les données dans la zone d'affichage

Côté UI

Pour afficher le tableau, nous utilisons la fonction `dataTableOutput`. Une zone va être créée pour afficher un tableau. Nous donnons à cette zone un identifiant en utilisant l'argument `outputId` . Cet identifiant est indispensable pour retrouver la zone côté serveur.

```
tabItems(
      # Read data
      tabItem(tabName = "readData",
              h1("Lecture des données"),
              fileInput("dataFile",label = NULL,
                        buttonLabel = "Browse...",
                        placeholder = "No file selected"),

              h3("Parameters"),

              # Input: Checkbox if file has header
              radioButtons(inputId = "header",
                           label = "Header",
                           choices = c("Yes" = TRUE,
                                       "No" = FALSE),
                           selected = TRUE, inline=T),

              # Input: Select separator ----
              radioButtons(inputId = "sep",
                           label = "Separator",
                           choices = c(Comma = ",",
                                       Semicolon = ";",
                                       Tab = "t"),
                           selected = "t", inline=T),

              # Input: Select quotes ----
              radioButtons(inputId = "quote",
                           label= "Quote",
                           choices = c(None = "",
                                       "Double Quote" = '"',
                                       "Single Quote" = "'"),
                           selected = "", inline=T),
              h3("File preview"),
              dataTableOutput(outputId = "preview")

      ),
```

Côté Server

Nous souhaitons à présent afficher de l'information. Du côté serveur, pour envoyer de l'information, la syntaxe commence quasiment toujours

par `output$` puis l'ID de la zone de sortie (ici notre tableau de prévisualisation avec l'id "preview"). Ce que nous souhaitons lui envoyer est un tableau. Nous utilisons donc la fonction `<- renderDataTable({ })`. Dans cette dernière fonction, nous allons lire le tableau qui va être renvoyé. Pour récupérer de l'information du côté UI, il faut utiliser la syntaxe suivante : `input$ID`. Par exemple, nous souhaitons récupérer le choix de l'utilisateur concernant le header : `input$header`.

```
output$preview <-  renderDataTable({

    req(input$dataFile)

    df <- read.csv(input$dataFile$datapath,
                   header = as.logical(input$header),
                   sep = input$sep,
                   quote = input$quote,
                   nrows=10
    )
  },  options = list(scrollX = TRUE , dom = 't'))
```

Si nous détaillons le code :

- **req(input$dataFile)** : bloque la suite du code si la zone d'import de fichier est vide
- **df <- read.csv()** : on stocke dans df la lecture du fichier
- **input$dataFile$datapath** : chemin d'accès au fichier importé
- **header = as.logical(input$header)** : récupération de la réponse de l'utilisateur pour savoir si présence ou absence d'un header. Le as.logical permet de convertir un TRUE ou FALSE en booléen.
- **sep = input$sep, quote = input$quote** : récupération du paramétrage de l'utilisateur pour le séparateur et les quotes. Ces informations sont données aux arguments de la fonction read.csv()
- **nrows=10** : Nous ne souhaitons pas lire tout le fichier. Seules les premières lignes sont nécessaires pour savoir si le tableau est lu correctement ou non. Nous lisons donc les 10 premières lignes.
- **options = list(scrollX = TRUE , dom = 't')** : Si le tableau a de nombreuses colonnes, cette option permet d'avoir un scroll horizontal

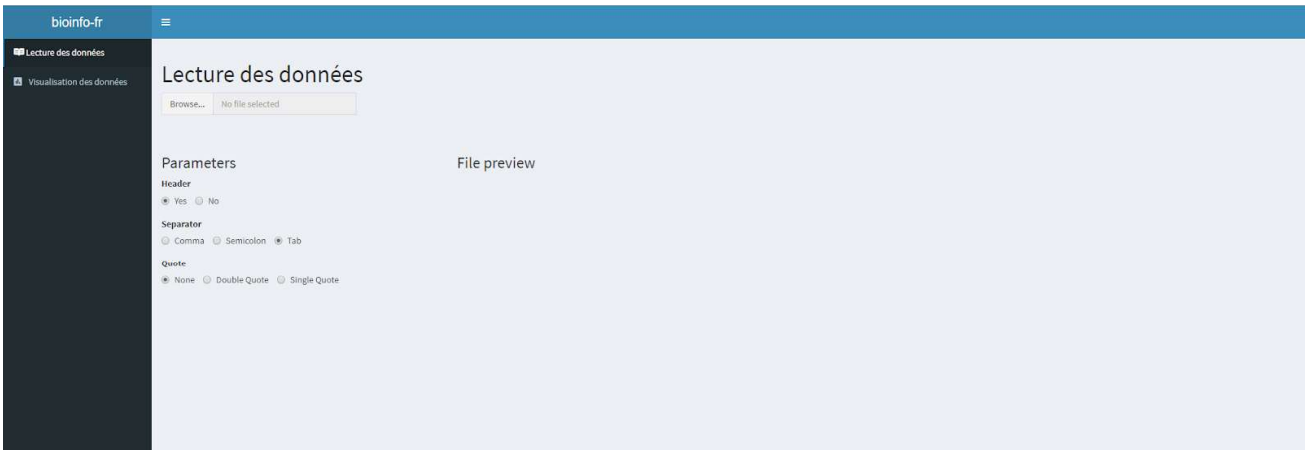
Vous pouvez maintenant tester sur un fichier texte contenant un tableau. Le changement de paramétrage a un effet direct sur la visualisation.

Visualisation de l'application

Organisation des éléments

Pour les connaisseurs de bootstrap, Shiny intègre son code. Pour les autres, il est possible d'organiser le contenu d'une page à l'aide d'une grille. La grille est composée de lignes (`fluidRow()`) elles-mêmes composées de 12 blocs. Nous allons placer les paramètres et la prévisualisation sur une même ligne. Nous souhaitons stocker les paramètres dans 3 blocs (`column(3,...)` : la colonne aura une taille de 3 blocs) et 9 blocs pour la prévisualisation (`column(9,...)`).

```
tabItem(tabName = "readData",
              h1("Lecture des données"),
              fileInput("dataFile",label = NULL,
                        buttonLabel = "Browse...",
                        placeholder = "No file selected"),

              fluidRow(
                column(3,
                       h3("Parameters"),

                       # Input: Checkbox if file has header
                       radioButtons(inputId = "header",
                                    label = "Header",
                                    choices = c("Yes" = TRUE,
                                                "No" = FALSE),
                                    selected = TRUE, inline=T),

                       # Input: Select separator ----
                       radioButtons(inputId = "sep",
                                    label = "Separator",
                                    choices = c(Comma = ",",
                                                Semicolon = ";",
                                                Tab = "t"),
                                    selected = "t", inline=T),

                       # Input: Select quotes ----
                       radioButtons(inputId = "quote",
                                    label= "Quote",
                                    choices = c(None = "",
                                                "Double Quote" = '"',
                                                "Single Quote" = "'"),
                                    selected = "", inline=T)
                ),
                column(9,
                       h3("File preview"),
                       dataTableOutput(outputId = "preview")
                )
              )
```



Visualisation de l'application

Bouton de lecture

Pour finir avec cette page, nous allons créer un bouton pour valider le paramétrage de la lecture du tableau. En cliquant sur ce bouton, l'ensemble du fichier sera lu. Nous ne réalisons pas une lecture dynamique comme précédemment. En effet, à chaque changement de paramètre, l'ensemble du fichier est relu. Si le fichier est gros, le temps de lecture sera long.

Côté UI

Nous ajoutons un `actionButton`. L'identifiant de notre bouton est "actBtnVisualisation".

```
[...]
actionButton(inputId = "actBtnVisualisation", label = "Visualisation",icon = icon("play") )
[...]
```

Pour une question esthétique, nous ajoutons un saut de ligne avant le bouton et nous mettons le bouton dans une division pour pouvoir le centrer :

```
[...]
tags$br(),
div(actionButton(inputId = "actBtnVisualisation", label = "Visualisation",icon = icon("play") ), align = "center")
[...]
```

Côté serveur

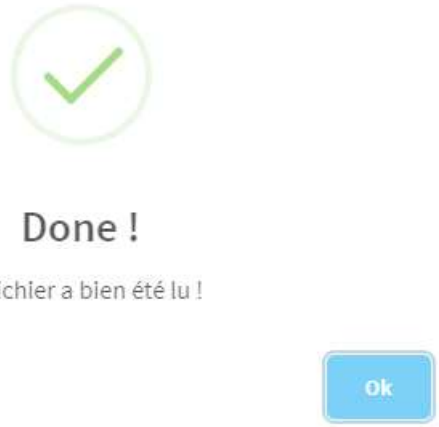
Lorsque que le bouton est cliqué, nous souhaitons à présent que le contenu du fichier soit stocké dans une variable. Il s'agit d'une variable particulière. Elle doit être visible par toutes les fonctions côté serveur et relancer toutes les fonctions qui l'utilisent si elle change. Il s'agit d'une variable réactive (`reactiveValues`). Si nous détaillons le code :

- Nous déclarons une reactiveValue avec comme nom `data`.
- Nous allons utiliser une fonction qui permet d'attendre une action particulière. Ici nous attendons que l'utilisateur clique sur le bouton. Une fois que le bouton a été cliqué, le code entre les { } sera exécuté. Ici, l'objectif sera de stocker le contenu du fichier importé dans la reactiveValue sous le nom `table` (data$table)

```
data = reactiveValues()

  observeEvent(input$actBtnVisualisation, {
    data$table = read.csv(input$dataFile$datapath,
                          header = as.logical(input$header),
                          sep = input$sep,
                          quote = input$quote,
                          nrows=10)
  })
```

Ainsi, à chaque clic du bouton, `data$table` sera mis à jour ainsi que toutes les fonctions qui l'utilise (ex : des graphiques).
Nous pouvons aussi ajouter un message pour confirmer la lecture du fichier. Nous utiliserons `sendSweetAlert` proposé dans le package shinyWidgets. La documentation est disponible ici : https://github.com/dreamRs/shinyWidgets .

```
observeEvent(input$actBtnVisualisation, {
    data$table = read.csv(input$dataFile$datapath,
                          header = as.logical(input$header),
                          sep = input$sep,
                          quote = input$quote,
                          nrows=10)
    sendSweetAlert(
      session = session,
      title = "Done !",
      text = "Le fichier a bien été lu !",
      type = "success"
    )
  })
```



Visualisation du message

Changement de page

Enfin, notre application étant composée de 2 pages, nous souhaitons changer de page une fois que le fichier est lu pour arriver sur la page de visualisation.

```
updateTabItems(session, "tabs", selected = "visualization")
```

Pour rappel, "tabs" est l'identifiant de notre sidebarMenu. Nous allons avec cette commande chercher dans la sidebarMenu la page qui a comme identifiant "visualization" et changer de page.

Visualisation

Exploration du tableau

Nous allons à présent afficher le tableau complet. Nous utilisons pour cela le package DT (https://rstudio.github.io/DT/). Il permet de rechercher, sélectionner ou trier les informations d'un tableau de données. Il faut pour cela créer une zone où sera affiché le tableau dans

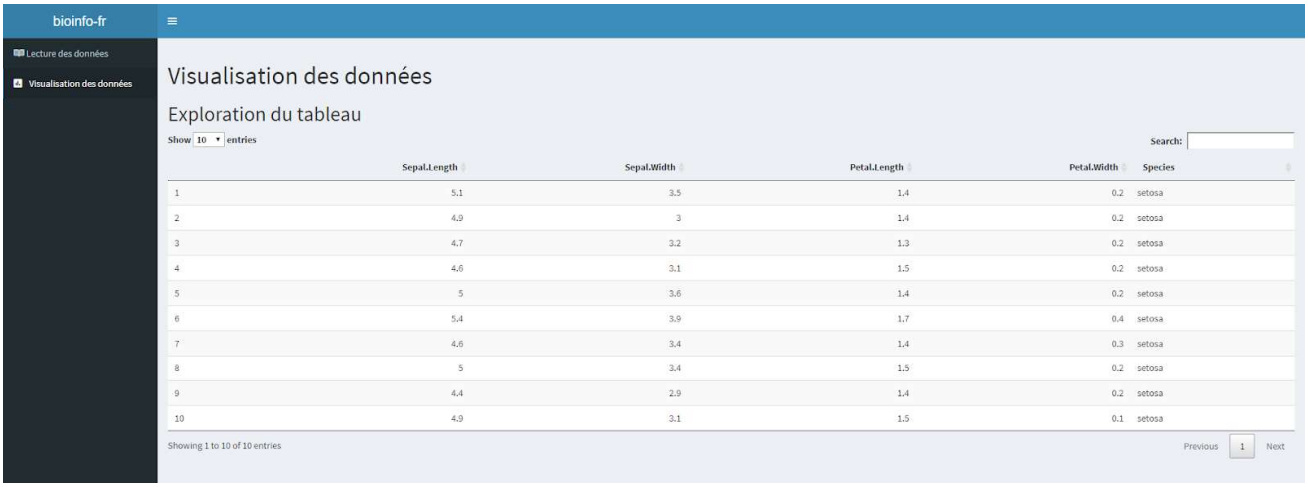
l'UI.

Côté UI

```
tabItem(tabName = "visualization",
              h1("Visualisation des données"),
              h2("Exploration du tableau"),
              dataTableOutput('dataTable')
      )
```

Puis du côté serveur, il ne reste plus qu'à envoyer le contenu de notre fichier dans ce tableau par le biais de la `reactiveValue`. Ainsi, le tableau sera automatiquement mis à jour si un nouveau fichier est lu.

```
output$dataTable = DT::renderDataTable(data$table)
```



Visualisation de l'application

Il est possible de faire de la mise en forme conditionnelle comme dans excel. Le code proposé par la suite est dépendant du tableau utilisé. En effet, nous allons cibler les colonnes d'intérêt par leur nom pour une question de lisibilité.

Voici une proposition de mise en forme conditionnelle de notre tableau (inspiré de l'exemple proposé dans la documentation du package DT).

- Histogramme des valeurs pour les colonnes Sepal.length et Petal.length
- Coloration par seuils multiples pour les colonnes Sepal.width et Petal.width (fond blanc écriture noire, fond rouge écriture blanche et fond rouge foncé écriture blanche)
- Coloration du fond en fonction de l'espèce pour la colonne espèce.

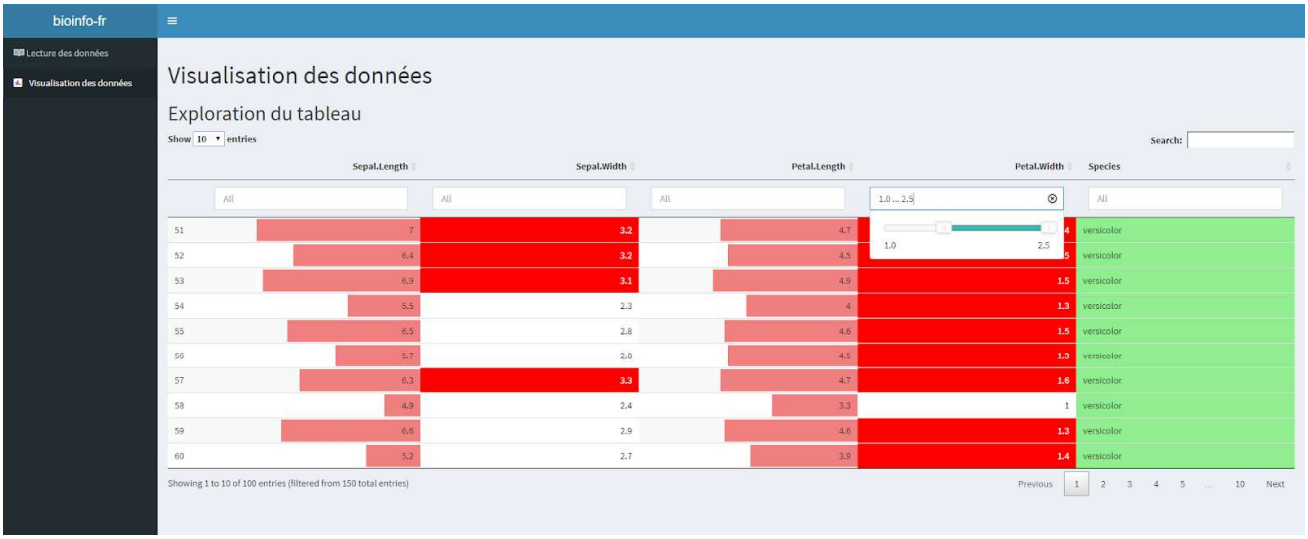
```
output$dataTable = DT::renderDataTable({
    datatable(data$table, filter = 'top') %>%
      formatStyle('Sepal.Length',
                  background = styleColorBar(data$table$Sepal.Length, 'lightcoral'),
                  backgroundSize = '100% 90%',
                  backgroundRepeat = 'no-repeat',
                  backgroundPosition = 'center'
      ) %>%
      formatStyle(
        'Sepal.Width',
        backgroundColor = styleInterval(c(3,4), c('white', 'red', "firebrick")),
        color = styleInterval(c(3,4), c('black', 'white', "white"))
      ) %>%
      formatStyle(
        'Petal.Length',
        background = styleColorBar(data$table$Petal.Length, 'lightcoral'),
        backgroundSize = '100% 90%',
        backgroundRepeat = 'no-repeat',
        backgroundPosition = 'center'
      ) %>%
      formatStyle(
        'Petal.Width',
        backgroundColor = styleInterval(c(1,2), c('white', 'red', "firebrick")),
        color = styleInterval(c(1,2), c('black', 'white', "white"))
      ) %>%
      formatStyle(
        'Species',
        backgroundColor = styleEqual(
          unique(data$table$Species), c('lightblue', 'lightgreen', 'lavender')
        )
      )
  })
```

Visualisation de l'application

Enfin, pour améliorer l'exploration, il est possible d'ajouter des filtres par colonnes. Pour les valeurs numériques, les données sont filtrées par un slider. Pour les colonnes contenant du texte, il y a deux possibilités :

- Peu de variabilité entre les éléments. Par exemple, la colonne Species ne contient que 3 éléments différents : setosa, versicolor et virginica. Dans ce cas, le filtre sera composé des éléments uniques de cette colonne qui seront cliquables. En les cliquant, toutes les lignes avec cet élément seront sélectionnées.
- Grande variabilité entre les éléments. Dans ce cas, une zone pour entrer du texte sera proposée. Le texte saisi sera recherché dans la colonne.

```
output$dataTable = DT::renderDataTable({
    datatable(data$table, filter = 'top') %>%
      [...]
```



Visualisation de l'application

Visualisation graphique

Nous allons créer de 4 façons différentes des graphiques et les afficher dans l'application shiny :

- des graphiques statiques
  - un plot de base avec R
  - un graphique avec ggplot2
- des graphiques dynamiques
  - Avec plotly
  - Avec google

Les graphiques seront représentés sur la même ligne avec une fluidRow et 4 colonnes (comme nous avons fait précédemment).

Graphique R

R propose une grande palette de graphiques de base. Cependant, il s'agit uniquement de graphiques statiques.

Côté UI

Il faut comme précédemment créer une zone pour indiquer où va être affiché le graphique. La fonction utilisée est plotOutput.

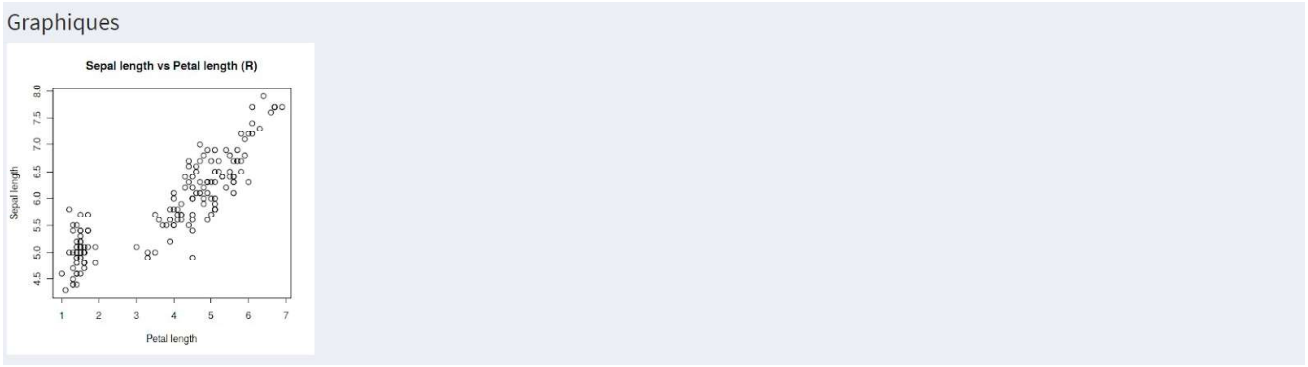
```
tabItem(tabName = "visualization",
                h1("Visualisation des données"),
                h2("Exploration du tableau"),
                dataTableOutput('dataTable'),
                h2("Graphiques"),
               fluidRow(
                  column(3,plotOutput("plotAvecR") )
                )
       )
```

Côté serveur

Nous allons pour ce graphique comparer la corrélation entre la taille des sépales et des pétales. Comme pour le tableau, la syntaxe est la suivante pour envoyer de l'information du côté UI : `output$ID_de_la_zone`. Pour envoyer un plot, nous utilisons la fonction renderPlot. Dans cette fonction, vous pouvez mettre n'importe quel graphique de R. Afin de mettre à jour automatiquement les graphiques, nous utilisons notre reactiveValue : `data`. Chaque fois que `data` changera, le plot sera généré de nouveau. Pour accéder au contenu du fichier lu qui est stocké dans la reactiveValue `data` sous le nom de `table`, nous utilisons de nouveau la syntaxe suivante : `data$table`. Il s'agit d'un dataframe (la lecture par read.csv2 renvoie un dataframe). Les colonnes sont donc accessibles par un $ puis le nom. Au final, pour obtenir le vecteur contenant les valeurs de longueur des pétales, nous utiliserons la syntaxe suivante : `data$table$Petal.Length`.

```
output$plotAvecR <- renderPlot({
    plot(data$table$Petal.Length,data$table$Sepal.Length,
         main = "Sepal length vs Petal length (R)",
         ylab = "Sepal length",
         xlab = "Petal length")
  })
```

Le paramétrage du plot est libre et n'est pas contraint par shiny.



Visualisation de l'application

Graphique par ggplot2

Ggplot2 est une librairie graphique de plus en plus utilisée. Elle propose des graphiques plus évolués que ceux de base dans R. Vous trouverez une documentation très bien faite ici : https://ggplot2.tidyverse.org/. Nous allons comparer les largeurs et les longueurs des sépales. Une coloration en fonction de l'espèce est proposée.

Côté UI

Comme toujours, nous allons créer une zone où sera affiché le graphique. La fonction utilisée est encore `plotOutput`.

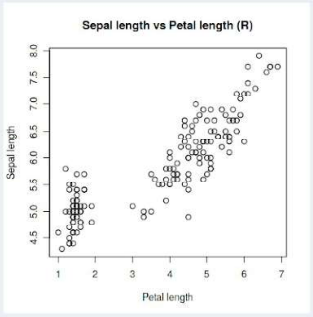
```
fluidRow(
                  column(3,plotOutput("plotAvecR")),
                  column(3, plotOutput("plotAvecGgplot2"))
                )
```

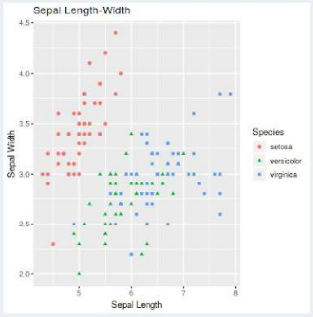
Côté serveur

Nous allons procéder de la même façon que précédemment. La différence est liée au contenu de la fonction `renderPlot`. Nous allons cette fois-ci utiliser les fonctions de ggplot2.

```
output$plotAvecGgplot2 <- renderPlot({
    ggplot(data=data$table, aes(x = Sepal.Length, y = Sepal.Width)) +
      geom_point(aes(color=Species, shape=Species)) +
      xlab("Sepal Length") +  ylab("Sepal Width") +
      ggtitle("Sepal Length-Width (ggplot2")
  })
```

Graphiques





Visualisation de l'application

Graphique Plotly

Plotly est un package de j'affectionne particulièrement. Il propose énormément d'outils préprogrammés (enregistrement de l'image, zoom, informations supplémentaires). De plus, il n'est pas exclusivement réservé à R. Il est possible de l'utiliser aussi dans des projets en JS et en python (aussi simple d'utilisation).

Côté UI

De nouveau, nous allons créer une zone pour afficher le graphique. Attention, nous changeons de fonction. Nous utiliserons cette fois plotlyOutput.

```
fluidRow(
                column(3, plotOutput("plotAvecR")),
                column(3, plotOutput("plotAvecGgplot2")),
                column(3, plotlyOutput("plotAvecPlotly"))
              )
```

Côté serveur

Je n'expliquerai pas ici la syntaxe pour réaliser un graphique avec Plotly. La documentation sur le site est extrêmement bien faite avec de très nombreux exemples (https://plot.ly/r/). Vous pouvez mettre n'importe quel graphique plotly dans la fonction. Ici, nous comparons la largeur et la longueur des pétales.

```
plot_ly(data = data$table, x = ~ Petal.Length, y = ~ Petal.Width, color = ~ Species) %>%
        layout(title = 'Petal Length-Width (plotly)',
               yaxis = list(title = "Petal width"),
               xaxis = list(title = "Petal length"))
```

Je vous invite lorsque vous lancerez l'application à survoler ce graphique. Il y a énormément d'informations disponibles et d'outils d'exploration.

Graphique Google

Pour finir, les graphiques de Google sont de plus en plus populaires et offrent un plus large choix de représentations que Plotly (calendrier, etc.). Ici, nous allons réaliser un histogramme de la largeur des pétales.

Côté UI

Nous créons de nouveau une zone pour afficher le graphique. La fonction utilisée est htmlOutput. Cette fonction est capable d'interpréter du code HTML venant du serveur. Si vous souhaitez écrire du HTML directement dans la partie UI, il vous suffit d'utiliser la fonction HTML (ex : HTML("<h1>Titre 1</h1>")) .

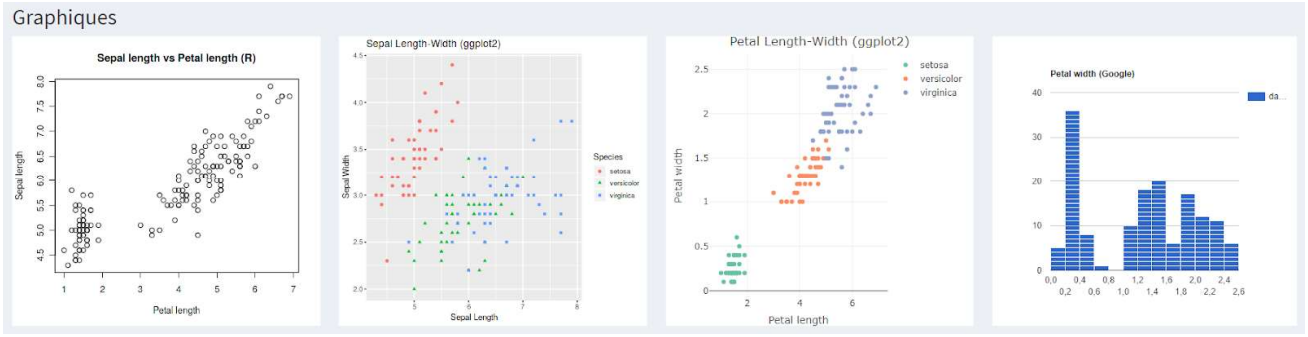
```
fluidRow(
                column(3, plotOutput("plotAvecR")),
                column(3, plotOutput("plotAvecGgplot2")),
                column(3, plotlyOutput("plotAvecPlotly")),
                column(3, htmlOutput("plotAvecGoogle"))
              )
```

Côté serveur

Pour les graphiques Google, nous utilisons les fonctions graphiques commençant par gvis et le rendu est fait avec la fonction `renderGvis`. Elles sont détaillées à la page suivante https://cran.r-project.org/web/packages/googleVis/vignettes/googleVis_examples.html .

```
output$plotAvecGoogle <- renderGvis({
      gvisHistogram(as.data.frame(data$table$Petal.Width),
                    options=list(title ="Petal width (Google)",
                                 height=400)
      )
  })
```

Visualisation de l'application



Visualisation du l'application

Gérer le tableau vide

En lançant l'application, si vous vous rendez sur la partie visualisation, vous trouverez plein d'erreurs. Ces erreurs sont la cause de l'utilisation d'une `reactiveValue`. En effet, lorsque rien n'a encore été lu, `data$table` est NULL (vide). Or toutes les fonctions que nous utilisons ne gèrent pas les NULL. Nous ajouterons pour le tableau et les graphiques un peu de code pour lui dire de renvoyer NULL si le tableau est vide.

```
if (!is.null(data$table)) {
    [représentation graphique ou le tableau]
} else {
    NULL
}
```

Interagir avec les graphique

Nous allons voir deux types d'interactions avec les graphiques pour illustrer la simplicité pour l'utilisateur d'interagir avec les données et les représentations :

- Sélectionner les données à afficher à l'aide du tableau
- Changer des paramètres graphiques sur le plot de base proposé par R (le premier graphique). Tous ces changements peuvent bien sûr être appliqués sur tous les graphiques.

Sélectionner les données à afficher à l'aide du tableau

Grâce à Shiny, il est possible de faire communiquer le tableau avec les graphiques. Nous profitons pour cela de la puissance du package DT qui génère le tableau. Les modifications que nous allons réaliser seront uniquement côté serveur. L'objectif est de récupérer les lignes qui sont affichées dans le tableau et de n'utiliser que ces lignes dans les graphiques. Comme précédemment, pour récupérer de l'information dans l'UI, il faut utiliser `input$ID_ZONE`. Nous souhaitons récupérer de l'information de notre tableau qui a comme identifiant `dataTable`. Ensuite, nous ajoutons `_rows_all` à la fin de l'ID pour obtenir les lignes. Ainsi, avec `input$dataTable_rows_all`, nous avons les lignes affichées dans le tableau. Il ne reste plus qu'à les sélectionner dans le vecteur de données. Le graphique est à présent dynamique.

```
output$plotAvecR <- renderPlot({
    if (!is.null(data$table)) {
      plot(data$table$Petal.Length[input$dataTable_rows_all],
           data$table$Sepal.Length[input$dataTable_rows_all],
           main = "Sepal length vs Petal length (R)",
           ylab = "Sepal length",
           xlab = "Petal length")
    } else {
      NULL
    }
  })
```

La même démarche est ensuite appliquée aux autres graphiques. Grâce aux filtres du tableau, nous avons ainsi la possibilité de sélectionner par les données numériques (longueur et largeur) et par l'espèce.

Changement de couleur pour le graphique de base R

L'objectif est de vous montrer une autre façon d'interagir avec les graphiques. En effet, il se peut que vous n'utilisiez pas de tableau dans votre application. De très nombreux exemples sont disponibles en ligne (ici par exemple : https://shiny.rstudio.com/gallery/). Nous allons

implémenter 4 changements sur ce graphique pour vous donner des exemples d'utilisation d'inputs :

- Changement de la couleur des points (avec l'utilisation d'un colour picker capable de gérer la transparence)
- Changement du type de point
- Changement de la taille des points
- Changement du titre

Côté UI

Pour plus de lisibilité lors de l'utilisation, nous avons changé la disposition des graphiques pour avoir sur une ligne le graphique R avec ses paramètres et sur une seconde les trois autres graphiques. Vous pouvez ainsi voir la simplicité de la réorganisation d'une page à l'aide du système de Grid.

```
tabItem(tabName = "visualization",
                h1("Visualisation des données"),
                h2("Exploration du tableau"),
                dataTableOutput('dataTable'),
                h2("Graphiques"),
                fluidRow(
                  column(4, plotOutput("plotAvecR")),
                  column(4, colourpicker::colourInput("colR", "Couleur graphique R", "black",allowTransparent = T),
                         sliderInput("cex", "Taille",
                                     min = 0.5, max = 3,
                                     value = 1,step = 0.2
                                    )),
                  column(4, selectInput(inputId = "pch", choices = 1:20, label = "Type de points",selected = 1),
                         textInput("title", "Titre", "Sepal length vs Petal length (R)") )
                ),
                tags$br(),
                fluidRow(
                  column(4, plotOutput("plotAvecGgplot2")),
                  column(4, plotlyOutput("plotAvecPlotly")),
                  column(4, htmlOutput("plotAvecGoogle"))
                )
        )
```

Pour faire entrer de l'information, nous avons besoin de 4 fonctions `input` : `colourInput` pour la couleur (du package colourpcicker), `sliderInput` pour la taille des points, `selectInput` pour le type de points et `textInput` pour le titre du graphique.

Côté serveur

Nous allons récupérer les entrées et les intégrer dans notre plot.

```
      plot(data$table$Petal.Length[input$dataTable_rows_all],
           data$table$Sepal.Length[input$dataTable_rows_all],
           main = input$title,
           ylab = "Sepal length",
           xlab = "Petal length",
           pch = as.numeric(input$pch),
           col = input$colR,
           cex = input$cex)
```

Visualisation dans l'application

Visualisation de l'application

Conclusion

Et voilà ! Vous avez réalisé une application complète capable de lire un fichier en fonction de paramètres et d'explorer ses données. Vous trouverez l'ensemble du code sur github ici :
https://github.com/bioinfo-fr/bioinfo-fr_Shiny . A travers ce post, nous avons vu comment rendre interactive l'exploration d'un tableau de données à l'aide de Shiny. Vos utilisateurs n'auront plus à voir votre code. Ils auront simplement à appuyer sur Run App. Il existe de nombreuses solutions de partage (https://shiny.rstudio.com/tutorial/written-tutorial/lesson7/). De nombreuses autres possibilités sont disponibles et pourront être détaillées dans d'autres articles (concaténérisation et intégration continue d'une application Shiny, par exemple).

Merci à mes relecteurs Aurélien C. et Ismaël P. pour leur aide !

Versions des outils utilisés

```
R version 3.5.1 (2018-07-02)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Debian GNU/Linux 9 (stretch)

Matrix products: default
BLAS: /usr/lib/openblas-base/libblas.so.3
LAPACK: /usr/lib/libopenblasp-r0.2.19.so

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C               LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8     LC_
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C                  LC_ADDRESS=C               LC_TELEPHONE=C             LC_

attached base packages:
[1] parallel  stats4    stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 [1] bindrcpp_0.2.2              shinycssloaders_0.2.0       shinyjs_1.0                 colourpicker_1.0
 [7] plotly_4.8.0                ggplot2_3.1.0               FactoMineR_1.41             DT_0.5
[13] DelayedArray_0.8.0          BiocParallel_1.16.5         matrixStats_0.54.0          Biobase_2.42.0
[19] IRanges_2.16.0              S4Vectors_0.20.1            BiocGenerics_0.28.0         shinydashboard_0.7.1

loaded via a namespace (and not attached):
 [1] bitops_1.0-6           bit64_0.9-7            RColorBrewer_1.1-2     httr_1.4.0             tools_3.5.1
 [8] rpart_4.1-13           Hmisc_4.1-1            DBI_1.0.0              lazyeval_0.2.1         colorspace_1.3-2
[15] tidyselect_0.2.5       gridExtra_2.3          bit_1.1-14             compiler_3.5.1         htmlTable_1.13.1
[22] checkmate_1.9.0        genefilter_1.64.0      stringr_1.3.1          digest_0.6.18          foreign_0.8-70
[29] pkgconfig_2.0.2        htmltools_0.3.6        htmlwidgets_1.3        rlang_0.3.0.1          rstudioapi_0.8
[36] jsonlite_1.6           acepack_1.4.1          dplyr_0.7.8            RCurl_1.95-4.11        magrittr_1.5
[43] leaps_3.0              Matrix_1.2-14          Rcpp_1.0.0             munsell_0.5.0          yaml_2.2.0
[50] MASS_7.3-50            zlibbioc_1.28.0        plyr_1.8.4             grid_3.5.1             blob_1.1.1
[57] miniUI_0.1.1.1         lattice_0.20-35        splines_3.5.1          annotate_1.60.0        locfit_1.5-9.1
[64] geneplotter_1.60.0     XML_3.98-1.16          glue_1.3.0             latticeExtra_0.6-28    data.table_1.11.8
[71] purrr_0.2.5            tidyr_0.8.2            assertthat_0.2.0       xfun_0.4               mime_0.6
[78] viridisLite_0.3.0      survival_2.42-3        tibble_1.4.2           AnnotationDbi_1.44.0   memoise_1.1.0
```

Bibliographie

[1] Winston Chang, Joe Cheng, JJ Allaire, Yihui Xie and Jonathan McPherson (2018). shiny: Web Application Framework for R. R package version 1.2.0.https://CRAN.R-project.org/package=shiny

[2] Winston Chang and Barbara Borges Ribeiro (2018). shinydashboard: Create Dashboards with 'Shiny'. R package version 0.7.1. https://CRAN.R-project.org/package=shinydashboard

[3] Victor Perrier, Fanny Meyer and David Granjon (2018). shinyWidgets: Custom Inputs Widgets for Shiny. R package version 0.4.4. https://CRAN.R-project.org/package=shinyWidgets

[4] Yihui Xie, Joe Cheng and Xianying Tan (2018). DT: A Wrapper of the JavaScript Library 'DataTables'. R package version 0.5. https://CRAN.R-project.org/package=DT

[5] Carson Sievert (2018) plotly for R. https://plotly-book.cpsievert.me

[6] H. Wickham. ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, 2016.

[7] Markus Gesmann and Diego de Castillo. Using the Google Visualisation API with R. The R Journal, 3(2):40-44, December 2011.

[8] Dean Attali (2017). colourpicker: A Colour Picker Tool for Shiny and for Selecting Colours in Plots. R package version 1.0. https://CRAN.R-project.org/package=colourpicker

[9] Aurelien Chateigner (2018). anyLib: Install and Load Any Package from CRAN, Bioconductor or Github. R package version 1.0.5. https://CRAN.R-project.org/package=anyLib